Harald Horchler (Hrsg.)

64'er

Assembler ist keine Alchemie

Der 64'er-Kurs zur Programmierung der Prozessoren 6502 / 6510 in Maschinensprache

Horchler

Assembler

Harald Horchler (Hrsg.)

Assembler ist keine Alchemie

Der 64'er-Maschinensprache-Kurs zur Programmierung der Prozessoren 6502 / 6510 in Maschinensprache

Skriptorium-Verlag
34326 Morschen

www.skriptorium-vd.de
info@skriptorium-vd.de

Lesefutter für Computerliebhaber

Verlagsanschrift:
Skriptorium-Verlag
Gartenstraße 4
34326 Morschen
Telefon: 0 56 64 / 93 99 46
Telefax: 0 56 64 / 93 99 46
Internet: www.skriptorium-vd.de
E-Post: info@skriptorium-vd.de

Satz und Layout: Harald Horchler

ISBN 3-938199-01-6

Printed in Germany

Verfasst in neuer deutscher Rechtschreibung

Horchler, Harald (Hrsg.): Assembler ist keine Alchemie. Der 64'er-Maschinensprache-Kurs zur Programmierung der Prozessoren 6502 / 6510 in Maschinensprache

© Copyright 2004 Skriptorium-Verlag. Alle Rechte vorbehalten. Jegliche Vervielfältigung von Texten, Tabellen, Grafiken oder Bildern, auch in elektronischer Form, bedarf der schriftlichen Genehmigung des Verlags.

Das Copyright für die abgedruckten Texte liegt bei der WEKA Computerzeitschriftenverlag GmbH & Co. KG, 85586 Poing (www.wekanet.de), die freundlicherweise die Genehmigung zur Wiederveröffentlichung erteilt hat. Ursprünglich wurden die Texte in verschiedenen Ausgaben der Zeitschrift »64'er« veröffentlicht.

Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Werk berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und der Markenschutzgesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürften.

Inhaltsverzeichnis

Einleitung 9

Assembler ist keine Alchemie – Folge 1 11
Einige Begriffsklärungen 12
BASIC kontra Assembler 13
Wie sag' ich's meinem Computer? 14
Wie funktioniert unser Computer? 14
Das Innenleben eines Mikroprozessors 15
Der Speicher unseres Computers: 17
Eine Straße mit 65536 Hausnummern 17

Assembler ist keine Alchemie – Folge 2 19
Wie sieht ein Assemblerprogramm aus? 20
Der zweite Assemblerbefehl: STA 21

Assembler ist keine Alchemie – Folge3 27
Eine Zauberformel der Assembler-Alchemisten:
INX, INY, INC, DEX, OEY, DEC 27
Befehle zum Zählen 27
Noch ein alchemistischer Zahlentrick 29
Das Geheimnis der negativen Binärzahlen 30
Komplement ist nicht kompliziert 31
Wie Variablen im Speicher stehen 31
Ein wirkungsvolles Zweiglein: BNE 33
Kein Widerspruch:Assembler-Programme langsamer machen 33

Assembler ist keine Alchemie – Folge 4 37
Herr Carry und der V-Mann 37
Der Computer rechnet: ADC, CLC 39
Noch mehr Rechnen: SBC, SEC 41
Ein Programmprojekt 42
Die Branch-Befehle 45

Assembler ist keine Alchemie – Folge 5 47
Die relative Adressierung 47
Zeropage-Adressierung 49
Die Vergleichsbefehle: CMP, CPX, CPY 52
Zeichenkodierung mit ASCII- und Commodore-ASCII-Kode 53
Die CHRGET-Routine 55

Assembler ist keine Alchemie – Folge 6 59
Einige Nachzügler: Die Befehle BIT, CLV, NOP
und TAX, TAY, TXA, TYA 60
So springen die Assembler-Alchemisten: JMP, JSR 62
Alles fließt: Fließkommazahlen 65

Die USR-Funktion 67
Der harte Kern: Noch mal Speicherfragen 68
Die Urzelle eines Programmprojekts 70
Einfache Befehle mit großer Wirkung 72

Assembler ist keine Alchemie – Folge 7 **75**
Wir stapeln 75
Der Stapel: Das Gedächtnis des Prozessors 76
Aktives Stapeln mit PHA, PLA, PHP, PLP, TSX und TXS 76
Sein oder Nichtsein: 78
Das Rätsel des Prozessorports 78
Die wandernden toten Briefkästen der Assembler-Alchemisten 79
Die indirekt-indizierte Adressierung 79
Die indiziert-indirekte Adressierung 82
Die ersten Kernal-Routinen 82

Assembler ist keine Alchemie – Folge 8 **89**
Die beiden ersten Interpreter-Routinen 94
Assembler-Befehle zum Beherrschen von Bits 95

Assembler ist keine Alchemie – Folge 9 **101**
Die restlichen Bit-Verschiebe-Operationen 101
Schneller Joystick 104
Die 16-Bit-Multiplikation 105
Die 16-Bit-Division 108
Das Programmprojekt wird fortgeführt 110
Die ROM-Bereiche als Datenquelle 113

Assembler ist keine Alchemie – Folge 10 **117**
Was sind Interrupts? 117
Das Unterbrechungssystem der CPU 6510/6502 119
Schlüssel zur Unterbrechungsprogrammierung:
CLI, SEI, RTI, BRK 119
Woher kommen die Unterbrechungs-Anforderungen? 122
Der VIC-II-Chip als Unterbrechungsquelle 122
Die beiden CIA-Bausteine als Unterbrechungsquellen 124
Lesen des Registers 124
Schreiben in das Register 124
Der IRQ-CIA 125
Der NMI-CIA 125
Die <RESTORE>-Taste und ein kleines Testprogramm 125

Assembler ist keine Alchemie – Folge 11 **127**
Der normale Verlauf eines IRQs 127
BRK-Unterbrechung 129
Was macht ein NMI? 130
Eigentlich keine Unterbrechung: RESET 132

Assembler ist keine Alchemie - Folge 12 **135**
Die Sache mit dem Modulstart 135
Nutzung der Unterbrechungen 136
Ein Programm zum VIC-II-IRQ 137

Assembler ist keine Alchemie - Folge 13 **143**
Die Timer der CIAs 144
Die Echtzeituhren 147

Ohne gutes Werkzeug geht es nicht: SMON - Folge 1 **156**
Was bietet SMON? 156
Assemblieren 157
Disassemblieren 158
Starten eines Maschinenprogramms (Go) 158
Memory-Dump 158
Registeranzeige 158
Exit 159
Das Programm 159

Ohne gutes Werkzeug geht es nicht: SMON - Folge 2 **162**
I/O-SET 162
LOAD 162
SAVE 163
PRINTER-SET 163
Ein bisschen Rechnerei 163
UMRECHNUNG DEZ-HEX 163
UMRECHNUNG HEX-DEZ 163
UMRECHNUNG BINÄR-HEX,DEZ 163
ADD-SUB 163
Programme auf dem Rangierbahnhof 164
OCCUPY (Besetzen) 164
WRITE 164
VARIATION 164
CONVERTIEREN (Verschieben eines 164
Programms mit Adressumrechnung) 164
Der Blick hinter die Kulissen 165

Ohne gutes Werkzeug geht es nicht: SMON - Folge 3 **168**
BASIC-DATA 168
KONTROLLE 169
FIND 169
Der Lösung ein Stück näher ... 171
Hinweis zum 16-Bit-Vergleich 172

Ohne gutes Werkzeug geht es nicht: SMON - Folge 4 **174**
Trace Stop 174
Trace Walk 175

Trace Break 177
Trace Quick 177
Wer steht wo? - Wegweiser durch SMON 178
Das »Gedächtnis« von SMON 179
SMON verschieben? Mit SMON! 179
SMON-Befehlsübersicht 181
Hinweise zur Befehlsübersicht 182

Ohne gutes Werkzeug geht es nicht: SMON - Folge 5 **184**
Die Befehle des Disk-Monitors 184
READ: R (Track Sektor) 184
MEMORY-DUMP: M 185
WRITE: W (Track Sektor) 185
ERROR: @ 185
EXIT: X 185
Reparatur eines gelöschten Files 185
Schützen eines Files 186
Schützen einer Diskette 186
Ändern des Diskettennamens oder der ID 187
Ändern eines Filetyps 187
Ändern der Startadresse eines Programms 187
Die Zusammenarbeit mit SMON 188
Die Ausgabe von Diskettenfehlern 188

Ohne gutes Werkzeug geht es nicht: SMON - Folge 6 **187**
Zum Schluss noch ein Tipp 188

Index I - Assembler ist keine Alchemie **189**
Index II - Ohne gutes Werkzeug geht es nicht: SMON **191**
Abbildungsverzeichnis **193**
Tabellen- und Programmverzeichnis **194**

Das Letzte aus dem Chip herausholen ...

... wollen alle Programmierer. Bei Computern mit Acht-Bit-Prozessoren stößt man jedoch schnell an deren Geschwindigkeitsgrenzen.

Daher sind für Programmierer zwei Dinge unverzichtbar: ein sauberer und möglichst Speicherplatz sparender Programmierstil und die Benutzung einer Programmiersprache, die der Prozessor möglichst ohne Umwege versteht.

Die Programmiersprache der Wahl heißt deshalb »Assembler«. Dieses Buch wendet sich an alle, denen das eingebaute BASIC des VC 20 oder C 64 zu langsam geworden ist - immerhin ist Assembler ungefähr 1000-mal schneller.

Dieser Kurs ist übrigens für alle Computer geeignet, die den Prozessor 6502 oder 6510 besitzen. Das sind neben Commodore-Rechnern auch Modelle von Apple, Atari und einigen anderen Herstellern.

Zusätzlich zum 64'er-Kurs »Assembler ist keine Alchemie« finden Sie im hinteren Teil des Buches noch die Bedienungsanleitung zu SMON, einem sehr leistungsfähigen und ausgereiften Assembler für den C 64.

Das Programm selbst und viele der angesprochenen Beispielprogramme können Sie unter folgender Internet-Adresse herunterladen:

www.skriptorium-vd.de/01/smon.d64

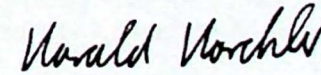
Wenn Sie auf einem »echten« C 64 programmieren und keine Möglichkeit zum Konvertieren der Datei auf eine 5,25-Zoll-Diskette haben, können Sie eine beim Verlag anfordern. Bitte senden Sie 4,95 € in Briefmarken (zum Beispiel 9 * 0,55 €) an:

Skriptorium-Verlag
SMON-Diskette
Gartenstraße 4
34326 Morschen
Deutschland

und vergessen Sie nicht, Ihre Adresse anzugeben.

Und nun viel Erfolg beim Erlernen von Assembler oder dem wieder Auffrischen alter Kenntnisse.

Viel Spaß beim Programmieren
wünscht allen Acht-Bit-Freunden



Assembler ist keine Alchemie
Folge 1

Erschienen in: 64'er 9/1984
Verantwortlicher Redakteur: Albert Absmeier
Autor: Heimo Ponnath

Viele Leser haben den Wunsch nach einem Maschinensprache-Kurs geäußert. Voilà, hier ist er.

Dieser Kurs wird Ihnen helfen, diese Sprache wesentlich leichter zu verstehen. Sowohl der 6502-Prozessor im VC 20 als auch der 6510-Prozessor im C 64 werden behandelt.

Vermutlich hat es Ihnen auch schon ab und zu in den Fingern gejuckt, wenn Sie von Wunderdingen gelesen haben, die man per Maschinensprache mit dem Computer machen kann.
Vielleicht haben Sie sogar schon mal nichts ahnend angefangen einzutippen, was Sie als Assemblerlisting sahen. Doch schon nach dem ersten »C000 LDA # $00« und <RETURN> weigerte sich der Computer mit einem lapidaren »SYNTAX ERROR«. Wieso, werden Sie sich gefragt haben, das ist doch nun die Sprache unserer Maschine, nämlich Maschinensprache, was habe ich falsch gemacht? Dann sind Sie sicherlich mal auf diese merkwürdigen BASIC-Programme gestoßen, in denen ein langer Wurm von DATA-Zeilen mit einem kleinen FOR-NEXT-POKE-Kopf vorne und einem SYS-Schwanz hinten enthalten ist, und die man BASIC-Lader nennt.

Sie haben fleißig Zahlen eingetippt - das Ganze hoffentlich sofort abgespeichert -, vorschriftsmäßig mit dem SYS-Befehl gestartet und auf einen scheintoten Computer geschaut, der nur noch durch Aus- und Einschalten wieder zu beleben war. Wenn Sie dann nach langer Fehlersuche den irrtümlich eingetippten Punkt durch ein Komma ersetzen (oft finden Sie auch keinen Fehler, denn bei langen DATA-Sequenzen schlägt der Druckfehlerteufel mit Vorliebe zu), werden Sie sich gefragt haben, warum in aller Welt dieses kleine Missgeschick den ganzen Computer abstürzen lässt.

Sie merken vermutlich schon, dass mir das alles und noch mehr (worüber ich schamhaft schweige) passiert ist. Die Konsequenz war, dass ich losging, um ein schlaues Buch zu erwerben. Aber merkwürdig: Damals tauchte der Begriff »Maschinensprache« in keinem Titel auf. Irgendwann begriff ich, dass Assembler und Maschinensprache irgend etwas miteinander zu tun haben.

Aber da fing das ganze Elend erst richtig an: Da gab es 6502-, Z 80-, 8080-, 8085-, 68 000-Assembler, da waren irgendwelche Schaltpläne - anscheinend, wie man wo was hinlötet, für mich als Nichtelektroniker eine Art moderner Kunst -, da war von CPU, Bussen, negativen Flanken und Zweiphasentakten die Rede.

Ich habe mich furchtbar über die Geheimsprache geärgert, die es dem Uneingeweihten verwehrt, etwas zu verstehen. Seither hat sich Einiges verändert. Die Geheimnisse sind keine mehr und ich werde Ihnen in dieser Serie ohne verschlüsselte Sprache die magischen Zirkel der Assembler-Alchemisten offenbaren.

Heute gibt es auch Bücher über »Maschinensprache auf dem Commodore 64«, und es sei Ihnen angeraten, ruhig auch das eine

oder andere durchzuarbeiten. Sie werden allerdings feststellen, dass die meisten davon gerade dort aufhören, wo es anfängt, spannend zu werden: bei der Benutzung von Routinen des Betriebssystems und des Interpreters. Deswegen soll der Schwerpunkt dieser Serie anders liegen:

Wir werden das notwendige Grundwissen über die Hardware nur ganz knapp behandeln und dann das Vokabular des 65XX-Assemblers kennen lernen. Den Hauptteil der Serie verbringen wir aber mit Dingen, über die es kaum Literatur gibt: nämlich, wie man für eine Unzahl von Programmieraufgaben nicht noch mal das Rad erfinden muss, weil es schon längst in unserem Computer existiert.

Bevor wir loslegen, will ich Ihnen noch etwas Literatur empfehlen:

* Wenn wir über den Speicheraufbau, das binäre und das hexadezimale Zahlensystem reden, ist die vorherige Lektüre der Serie »Reise durch das Wunderland der Grafik« nützlich, die diese Themen in den Folgen 1 und 2 (64'er 4/1984 und 5/1984) grundlegend behandelt hat.
* Als Nachschlagebuch sehr wertvoll ist das Buch von Rodney Zaks: Programmierung des 6502. Das ist gewissermaßen ein Klassiker.
* Später wird Ihnen dieses Buch fast unentbehrlich vorkommen: Babel, R.; Krause, M.; Dripke, A.: Systemhandbuch zum Commodore 64 (und VC 20), München 1983

Weitere Literaturempfehlungen werde ich Ihnen von Fall zu Fall geben. Gerade zu unserem Computer erscheint fast jeden Monat ein neues Buch und es ist nicht einfach, die Spreu vom Weizen zu trennen.

Einige Begriffsklärungen

Zunächst einmal muss ich Sie enttäuschen: Ich glaube kaum, dass Sie mit Ihrem Computer je einmal in Maschinensprache verkehren werden! Maschinensprache - das ist die einzige, die der Computer direkt versteht, das sind vorhandene oder nicht vorhandene Stromimpulse oder Magnetisierungszustände, die bei unserem Computer durch Acht-Bit-Binärzahlen auszudrücken sind.

Was wir mit unserem Computer reden werden, ist Assembler. Mit dem Computer sprechen soll heißen: mit dem Gehirn unseres Computers, dem Prozessor, oft auch CPU (von Central Processing Unit = zentraler Arbeitsbaustein) genannt, verkehren, ihm also Befehle geben. Solche CPUs werden bei verschiedenen Firmen hergestellt, sind daher unterschiedlich aufgebaut und auch unterschiedlich ansprechbar.

Ein weit verbreiteter Prozessortyp ist der 6502, der das Gehirn des C 64 und auch des VC 20 ist. Genau genommen ist das Gehirn des C 64 allerdings der 6510, ein mit dem 6502 fast identischer Prozessor. Auf den kleinen Unterschied werden wir noch zu sprechen kommen. Beide (6502 und 6510) sind in 6502-Assembler zu programmieren, und wenn wir diese Sprache sprechen, sind für uns alle 6502-Computer zugänglich: Commodore, Apple, Atari und einige andere.

Nun wissen Sie aber immer noch nicht, was Assembler eigentlich ist. Das englische Wort »assemble« heißt auf deutsch etwa »montieren, zusammenstellen«.

Es handelt sich also um eine Programmiersprache, und weil sie sehr eng am Computer orientiert ist, spricht man von einer »maschinenorientierten« Programmsprache - im Gegensatz zu »problemorientierten« Programmsprachen wie BASIC, Pascal, Cobol etc., die - so sollte es jedenfalls sein - auf jedem Computertyp gleich aussehen.

Ein Assembler ist aber noch etwas anderes, nämlich ein Software-Instrument, das einen in Assembler geschriebenen Befehl in die Maschinensprache übersetzt. Man spricht dabei vom Vorgang des »assemblierens«.

Das Umgekehrte leistet ein Disassembler, welcher uns Maschinensprache durch Rückübersetzung lesen hilft. Um die Verwirrung noch etwas zu steigern, sage ich Ihnen auch

noch, was ein Monitor ist: In diesem Zusammenhang ist kein Bildschirmgerät damit gemeint, sondern ebenfalls ein Software-Instrument, das den Einblick in die Register und Speicher des Computers gewährt.

Damit Sie nun den Überblick noch völlig verlieren, sei abschließend zu diesem Sprachenwirrwarr noch erzählt, dass Software-Pakete, die sowohl Assembler als auch Disassembler als auch Monitor enthalten und noch eine Menge anderer brauchbarer Dinge, oft als »Assembler« angeboten werden. Das ist ein alter Trick der Alchemisten: verschiedenen Dingen den gleichen Namen zu geben!

BASIC kontra Assembler

Um das Nachfolgende deutlich zu machen, schalten Sie bitte Ihren Computer an und tippen die beiden folgenden Programme ein, die beide genau dasselbe tun: das obere Viertel des Bildschirms mit dem Buchstaben A zu füllen (beim VC 20 ist es die obere Hälfte). Zunächst einmal in BASIC:

```
10 FOR I = 1024 + 255 TO 1024 STEP -1
20 POKE I,1 : POKE 1 + 54272,14
30 NEXT I
```

Für den VC 20 (Grundversion und Drei-KByte-Erweiterung) ist zu setzen: An Stelle von 1024 jetzt 7680, statt 54272 jetzt 30208 und statt 14 die 6. Wenn Sie mehr als die 6,5 KByte im VC 20 haben, dann setzen Sie statt 1024 jetzt 4096, statt 54272 jetzt 34304 und ebenfalls statt 14 die 6.

Das Programm braucht 55 Byte + 7 Byte für die Variable I, macht zusammen 62 Byte Speicherplatz. Es geht ganz schnell, und wenn Sie es schaffen, können Sie ja mal mitstoppen, wie lange es von »RUN« bis »READY« braucht: zirka 4 Sekunden.

Jetzt dasselbe in Assembler. Weil wir aber noch nicht soweit sind, erst mal als BASIC-Lader, der uns das Programm in den Speicher bringt (wir kommen dazu gleich noch). Geben Sie also »NEW« ein und dann:

```
10 FOR I = 7000 TO 7000 +16
20 READ A : POKE I,A : NEXT I : END
30 DATA 160,255,162,14,169,1,153,255,3,
138,153,255,215,136,208,244,96
```

Beim VC 20 geben Sie bitte statt der 14 (Zeile 30, vierte Zahl) eine 6 ein. Starten Sie den BASIC-Lader mit »RUN« und nach dem »READY« geben Sie »NEW« und »CLR« ein: Wir brauchen den BASIC-Lader nicht mehr.

Ab Speicherstelle 7 000 steht jetzt unser Assemblerprogramm als Maschinenkode. Dass es wirklich dasselbe tut wie das BASIC-Programm, erfahren Sie durch »SYS 7000«.

Da hatten Sie vermutlich gar keine Zeit mehr, auf die Stoppuhr zu drücken! (5,4 Millisekunden etwa dauert das ohne die Zeit, die der BASIC-Interpreter für den Befehl »SYS« benötigt). Außerdem braucht das Programm 17 Byte Speicherplatz.

Genau das ist es, was die Assemblerprogrammierung so reizvoll macht: Der Speicher fasst mehr an Programm und die Ausführung des Programms geht fast 1 000-mal so schnell! Dazu kommen natürlich noch einige andere Kriterien, denn viele Probleme sind zum Beispiel in BASIC nicht lösbar, sondern nur mit dem vielseitigeren Assembler.

Unser Computer ist darauf vorbereitet, dass wir ihn in BASIC ansprechen. Er enthält im Normalfall sofort nach dem Einschalten ein stets präsentes Übersetzungsprogramm, den Interpreter, welcher unsere BASIC-Anweisungen für ihn verständlich interpretiert. Auch das ist ein Unterschied zu Assemblerprogrammen: Ist ein solches Programm erst einmal assembliert (also als Maschinensprache im Speicher vorhanden), braucht man kein Übersetzungsprogramm mehr.

BASIC-Programme dagegen müssen bei jedem Durchlauf von vorne bis hinten ständig übersetzt werden, sie laufen nicht ohne vorhandenen Interpreter. Wie so ein Interpreter im Prinzip arbeitet und was ihn von einem so genannten »Compiler« unterscheidet, sollten Sie mal in der 64'er 4 / 1984 im Artikel von M. Törk über seinen Strubs-Pre-

compiler nachlesen. Dort sehen Sie dann auch, dass ein Compiler zwar ein BASIC-Programm enorm beschleunigen kann, aber bei weitem nicht an die Geschwindigkeit reiner Assembler-Programme heranreicht, vom Speicherplatzbedarf ganz zu schweigen.

Wie sag' ich's meinem Computer?

Leider haben weder der C 64 noch der VC 20 einen Assembler implementiert (Sie merken, dass jetzt von dem Software-Paket die Rede ist!). Es gibt einen etwas mühseligen Weg, dieses Handikap zu umgehen: den BASIC-Lader. Wie ist also der Weg, mit einem solchen Lader eigene Maschinenprogramme in den Computer zu bekommen?

* Erstellen des Assembler-Programms. Das zu lernen ist die Hauptaufgabe der Serie. Das Ergebnis wird eine Kette von Befehlen sein, zu denen zum Beispiel der Befehl RTS gehört.
* Jedem Befehl in Assembler entspricht in Maschinensprache ein Binärkode in einer Speicherstelle. Diese Kodes sind in Listen nachschlagbar: RTS entspricht dem Binärkode 0110 0000.
* Der Kode muss in eine Speicherstelle eingegeben werden. Das geschieht von BASIC aus mit dem POKE-Befehl. Weil aber BASIC keine Binärzahlen kennt, muss der Kode ins Dezimalsystem umgerechnet werden. Glücklicherweise sind in den Tabellen meist schon die Kodes als Dezimal- oder wenigstens als Hexadezimalzahlen enthalten. RTS ist dezimal 96 (oder hex.60, das auch $60 geschrieben werden kann). Man POKEt nun an die richtige Adresse den Wert 96, also zum Beispiel »POKE 7016,96«.
* Auf diese Weise wird Byte für Byte in der Programmabfolge verfahren. Das reine POKEn geschieht dann eben in der Form wie im oben gezeigten BASIC-Lader.

Mühsam, mühsam! Auch kann man leider nur mit dem PEEK-Kommando nachsehen, was denn nun im Speicher steht (PEEK (7016) gibt uns den Wert 96, entsprechend RTS).

Ein anderer Weg ist der Kauf eines Assembler-Moduls oder einer entsprechenden Programm-Kassette oder -Diskette.

Sehr preiswert ist es sicherlich, in Fachzeitschriften Umschau nach abgedruckten Assemblern zu halten. Sehr gut finde ich beispielsweise den Assembler von U. Roller in Chip 1 / 1984, aber auch in der 64'er wird während des Verlaufs dieser Serie ein gut verwendbares Programm vorgestellt.

Assembler (die Software-Pakete) gibt es in den unterschiedlichsten Ausführungen. Es gibt beispielsweise Direkt-Assembler, die jede Programmzeile sofort nach dem <RETURN> assemblieren, aber auch Zwei-Pass-Assembler, bei denen dies erst nach Abschluss des Programms insgesamt durch einen Befehl (zum Beispiel »ASSEMBLE«) geschieht. Bei einigen kann man (ähnlich wie bei BASIC mit »REM«) Kommentare anfügen, bestimmten Programmstellen Namen geben (so genannte »LABELS«), ganze Programmabschnittte mit einem Merknamen aufrufen (so genannte »MAKROS«) und so weiter. Was Sie für sich bevorzugen, bleibt natürlich Ihnen überlassen.

Die in dieser Serie geschriebenen Programme werden auf diese schönen Erleichterungen verzichten, es wird sozusagen der nackte Assembler verwendet, und solange in dieser Zeitschrift noch kein Software-Paket dafür veröffentlicht wurde, werde ich jedes Programm, das wir zusammen entwickeln, als BASIC-Lader angeben.

Was Sie aber außer dem reinen Assembler noch brauchen, sind ein Disassembler und ein Monitor (ich habe schon erklärt, welchen ich meine), damit wir unseren Computer (fast) immer im Griff haben.

Wie funktioniert unser Computer?

Weil das Programmieren in Assembler einen viel engeren Kontakt zu technischen Einzelheiten unseres Computers erfordert, ist es

notwendig, ein wenig über diese Innereien und ihre Funktion zu wissen. Sehen Sie sich dazu bitte Abbildung 1 an.

Da sehen wir zunächst unseren Mikroprozessor, der meist eine Menge Funktionen in sich vereinigt (dazu kommen wir noch). Im Prinzip ist das unsere CPU (Zentraler Arbeitsbaustein). Der Prozessor steht über eine Reihe von Leitungen mit dem Rest des Computers in Verbindung. Diese Leitungen werden im Fachjargon »Busse« genannt.

Da ist zunächst einmal der so genannte Adressbus, auf dem transportiert werden, die der Prozessor erzeugt, und die die Herkunft oder auch das Ziel von Daten festlegen, die über den Datenbus laufen.

Der Datenbus kann 8-Bit-Daten transportieren, und zwar schreibend oder lesend, also zum Beispiel vom Prozessor zum RAM (schreibend), vom RAM zum Prozessor (lesend) und so weiter. Außerdem gibt es da noch einen Steuerbus, der verschiedene Synchronisationsaufgaben durchführen hilft.

Links vom Prozessor ist ein Taktgeber angedeutet. Damit nichts durcheinander gerät, läuft alles im Computer sozusagen im Gleichschritt.

Diese Uhr ist gewissermaßen der Trommler, den Sie vielleicht von den alten Ruder-Galeeren kennen. Dann sehen Sie rechts einen ROM-Bereich, also einen Nur-Lese-Speicher (Read Only Memory). Dass man hier nur herauslesen kann, ist durch den Pfeil zum Datenbus gekennzeichnet.

Doppelpfeile finden wir aber beim RAM (Random Access Memory), einem Speicher für beliebigen Zugriff, also lesend und schreibend, und bei den Ein- und Ausgabebausteinen, die den Kontakt des Computers mit der übrigen Welt erlauben, also auch mit uns. Dieses Aufbauprinzip finden wir bei allen 8-Bit-Computern.

Das Innenleben eines Mikroprozessors

Um es gleich noch mal zu sagen: Was hier erzählt wird, ist nicht dazu geeignet, Elektronik-Freaks den totalen Durchblick zu geben. Wenn Sie das aber gerne möchten, dann sehen Sie sich zum Beispiel die Blockschaltbilder im »Programmer's Reference Guide« für den Commodore 64 auf Seite 404 oder im »MOS-Hardware-Handbuch« auf Seite 34 an.

Auch Rodney Zaks in dem anfangs schon erwähnten Buch »Programmierung des 6502« ist zu empfehlen. Er hat sich viel Mühe gegeben, sich verständlich auszudrücken. Mir kommt es nur auf den allgemeinen Überblick an. Den sollen Sie bekommen, wenn wir uns jetzt Bild 2 betrachten.

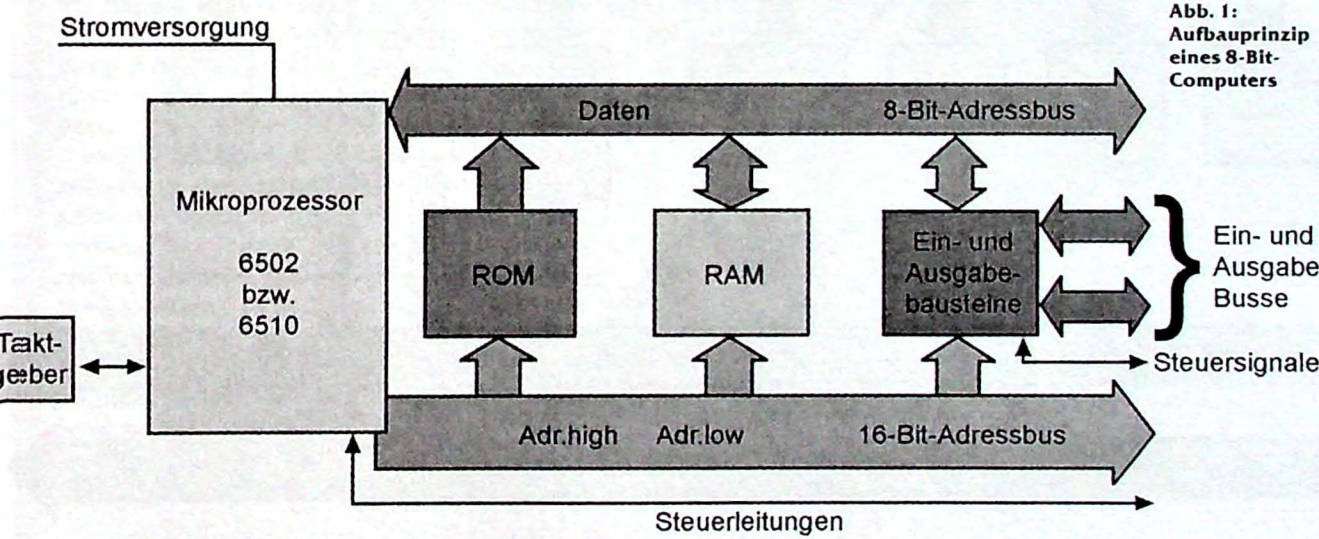


Abb. 1: Aufbauprinzip eines 8-Bit-Computers

Da sehen Sie zunächst als Herzstück des Prozessors die ALU (Arithmetic Logical Unit), also den arithmetisch-logischen Baustein. Die ALU hat die Fähigkeit, Rechenoperationen auszuführen mit Daten, die sie über den Datenbus und normalerweise vom Akkumulator erhält. Das Ergebnis wird ebenfalls im Akkumulator abgelegt (daher auch der Name: von akkumulieren, etwa ansammeln).

Der Akkumulator ist das Register, das uns als Programmierer am häufigsten beschäftigen wird. Er ist die Sammel-, aber auch die Verteilerstelle für fast alle Daten, die wir hin und her schieben wollen. Sowohl beim Akku (so werde ich, mit der Hoffnung auf Ihr wohl wollendes Verständnis, den Akkumulator künftig bezeichnen) als auch bei allen anderen Registern ist die höchste Zahl, die darin bearbeitet werden kann, 255 (binär 1111 1111).

Nahezu ebenso oft wie den Akku werden wir die beiden so genannten Index-Register X und Y benutzen. Warum man sie Index-Register nennt, sehen Sie im Verlauf der Serie noch.

Als Nächstes zum Prozessor-Statusflaggen-Register (hier P genannt). Man findet hier angezeigt, ob eine Rechenoperation ein negatives Ergebnis hatte, ob eine Null aufgetaucht ist oder ob ein Übertrag stattgefunden hat. Auch dieses Register wird uns noch häufig begegnen.

Das Stapelregister, auch Stackpointer (Stapelzeiger) genannt, gibt uns Auskunft über den Füllungsgrad eines 256 Byte großen speziellen Speichers, der vom Prozessor direkt verwaltet wird. Auch damit werden wir noch oft zu tun haben.

Schließlich kommen wir zur vorhin erwähnten Ausnahme, zum Programmzähler (PCL, PCH). Das ist ein 16-Bit-Register, das sich aus zwei 8-Bit-Registern (PCL für das LSB und PCH für das MSB) zusammensetzt und daher alle 65 535 Speicherplätze ansprechen kann. Hier ist immer die Adresse des nächsten abzuarbeitenden Befehls enthalten. Ich will an dieser Stelle nicht in die Einzelheiten der Befehlsabarbeitung einsteigen (das können Sie auch bei Rodney Zaks nachlesen, wenn Sie's genau wissen wollen). Es soll nur gesagt sein, dass sich die

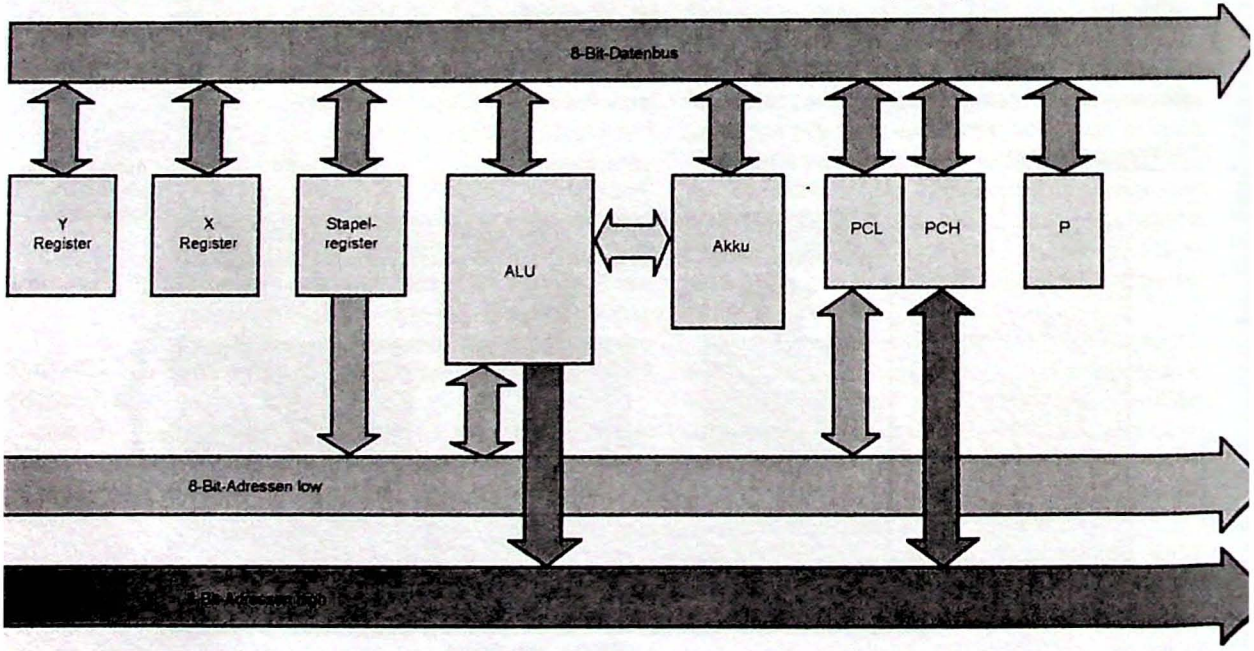


Abb. 2: Aufbauschema eines 6510-Prozessors

Verarbeitung in drei Schritte unterteilen lässt:

a) den nächsten Befehl holen
b) den Befehl dekodieren
c) den Befehl ausführen

Zu c) ist noch zu sagen, dass es Befehle gibt, die der Prozessor ohne weitere Angaben ausführen kann. Für andere müssen erst noch weitere Daten aus dem Speicher geholt oder dort abgelegt werden. Deswegen brauchen die Befehle unterschiedliche Zeiten zur Ausführung. Die Zeit wird als Anzahl von so genannten Taktzyklen in den Befehlstabellen angegeben.

Unser Computer hat eine Taktfrequenz von rund 1 MHz, was bedeutet, dass ein Taktzyklus etwa eine Mikrosekunde (10^{-6} s) dauert. Auf diese Weise wurde die Zeitdauer für unser kleines Demonstrationsprogramm zu Anfang berechnet. Auch das werden Sie noch lernen.

Der Speicher unseres Computers: eine Straße mit 65 536 Hausnummern

Diese Serie ist für den VC 20 und den C 64 geschrieben. Den Speicheraufbau des Commodore 64 finden Sie in der 64'er 4 / 1984 ab Seite 119. Deswegen soll hier nur der des VC 20 gezeigt werden. Man muss beim VC 20 zwei Konfigurationen unterscheiden - sehr zum Leidwesen der Benutzer. In Abbildung 3 ist die Aufteilung gezeigt, die in der Grund- und der um 3 KByte erweiterten Version vorliegt.

In Abbildung 4 sehen Sie die Speicheraufteilung, die bei mehr als 6,5 KByte eingestecktem Speicher gültig ist.

Wenn Sie die VC 20-Speicherarchitekturen mit der des C 64 vergleichen, werden Sie eine Reihe von Unterschieden feststellen. Genau besehen gibt es an den wichtigen Punkten aber eine Menge Gemeinsamkeiten! Der VC 20 kennt nur Speicher-Häuser mit Erdgeschoss - im Gegensatz zum C 64, wo manche Bereiche sogar zwei Etagen haben (soll heißen: mehrfach belegt sind). Durch die Eigenart des C 64 aber, im Normalfall das BASIC-ROM, die Ein- und Ausgabebausteine und das Betriebssystem eingeschaltet zu haben, kann man ihn eigentlich genauso behandeln wie einen VC 20, bei dem die genannten ROM-Bausteine - und zwar das BASIC-ROM - um 8 KByte verschoben sind. Die Unterschiede der ROM-Inhalte sind fast vernachlässigbar. Wir werden im Einzelfall darauf zu sprechen kommen. Bei den Ein- und Ausgabebausteinen gibt es allerdings größere Unterschiede.

Die Seiten 0 bis 3 (eine Seite oder auch Page enthält 256 Byte, und man zählt oft auch in diesen Seiten, wenn vom Speicher die Rede ist) sind sich ebenfalls sehr ähnlich, und die wenigen Unterschiede werden uns ebenfalls noch beschäftigen.

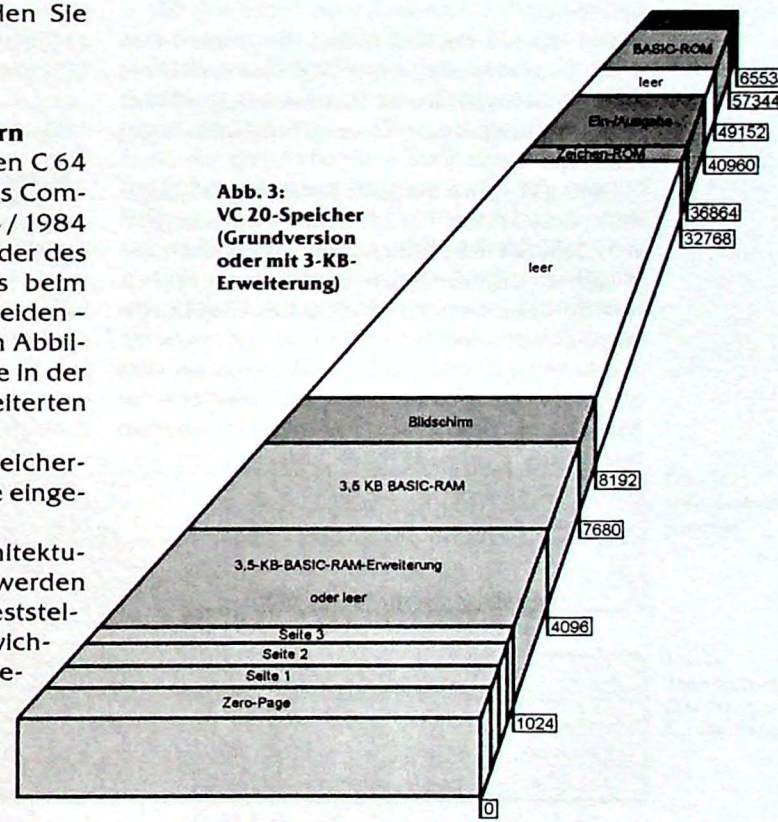


Abb. 3: VC 20-Speicher (Grundversion oder mit 3-KB-Erweiterung)

Tab. 1: BASIC-Start- und Endadressen beim VC 20 mit unterschiedlichem Speicherausbau

Grundversion	BASIC-Start:	4096	BASIC-Ende:	7679
+ 3-KB-Erweiterung	BASIC-Start:	1024	BASIC-Ende:	7679
+ 8-KB-Erweiterung	BASIC-Start:	4608	BASIC-Ende:	16383
+ 16-KB-Erweiterung	BASIC-Start:	4608	BASIC-Ende:	24575
+ 24-KB-Erweiterung	BASIC-Start:	4608	BASIC-Ende:	32767

Der Bildschirm liegt bei der Grundversion und der mit der 3-KByte-Erweiterung von 7680 bis 8191, in der Version mit mehr als 6,5 KByte von 4096 bis 4607 und beim C 64 von 1024 bis 2047.

Der Bildschirmfarbspeicher liegt - bei gleicher Reihenfolge - von 37888 bis 38399 bzw. von 38400 bis 38911 und von 55296 bis 56295.

Der BASIC-RAM-Bereich beginnt beim C 64 im Normalfall bei 2048 und endet bei 40959. Beim VC 20 ist das natürlich wieder von der jeweiligen Erweiterung abhängig (Tab. 1).

Dies gilt - wie Sie leicht aus Abbildung 4 ersehen können - auch dann, wenn zu den 8- / 16- / 24-KByte-Erweiterungen noch die 3-KByte- und die 8-KB-Erweiterung im hohen Speicherbereich (40960 bis 49151) verwendet werden.

Die letztgenannten Adressenbereiche sind dann gut als geschützte RAM-Bereiche für Maschinensprache zu verwenden, ebenso wie beim C64 der Speicherabschnitt von 49152 bis 53247.

Damit schließen wir zunächst mal den Hardware-Überblick ab. In der nächsten Folge stelle ich Ihnen dann die ersten Assembler-Befehle vor, und wir beginnen zu programmieren. Sie besitzen nun den ersten Grad der Assembler-Alchemisten.

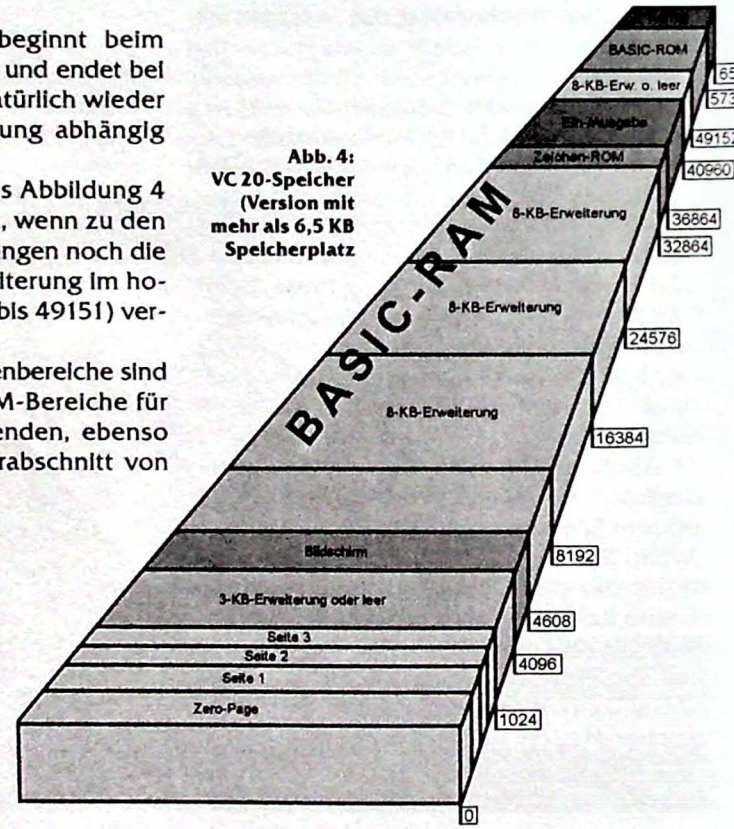


Abb. 4: VC 20-Speicher (Version mit mehr als 6,5 KB Speicherplatz

Assembler ist keine Alchemie
Folge 2

Erschienen in: 64'er 10/1984
Verantwortlicher Redakteur: Albert Absmeier
Autor: Heimo Ponnath

Bisher haben wir uns mit dem Innenleben unserer Computer auseinander gesetzt und die wichtigsten Teile der Hardware kennen gelernt. Jetzt kommen wir wie versprochen zur Software, nämlich zum Assembler.

Sollten Sie einen Assembler haben, dann schalten Sie ihn jetzt bitte ein. Falls Sie keinen haben, schaffen wir Abhilfe: Wir werden in den nächsten Ausgaben ein komplettes Software-Paket (Assembler, Disassembler und Monitor) veröffentlichen.

Dieser Assembler wird uns dann den ganzen Kurs hindurch begleiten. Meistens meldet sich der Assembler mit einer Registeranzeige.

Sollte Ihrer das nicht tun, dann müssen Sie wohl noch den Monitor anschalten oder speziell einen Befehl für die Registeranzeige eingeben (häufig ist das ein R). Jedenfalls wird nun auf dem Bildschirm der Inhalt der Register angezeigt (Tabelle 2). Die angezeigten Werte sind Beispiele, wie sie beim C 64 auftreten können. PC ist der Programmzähler, der immer auf den nächsten zu holenden Befehl zeigt. (Der Wert $E147 rührt vom SYS-Aufruf, mit dem ich meinen Assembler starte). IRQ zeigt uns an, auf welche Adresse der so genannte Interrupt-Vektor gestellt ist. Das ist das Byte-Paar 788 (LSB) und 789 (MSB). Im Normalfall zeigt es auf den Wert $EA31.

Die nächsten acht Angaben beziehen sich auf das Prozessorstatusregister, das wir in der letzten Folge P genannt haben. Die Bedeutung der einzelnen »Flaggen« zeigt Ihnen Tabelle 3.

AC ist der aktuelle Inhalt des Akkus. XR zeigt an, was im X-Register, und YR, was im Y-Register enthalten ist. SP (von »Stackpointer« = Stapelzeiger) gibt uns Auskunft über den freien Platz im Stapelregister. Damit wissen wir genau, was in diesem Moment in unserem Computer vorgeht. So fremd Ihnen das alles im Augenblick noch vorkommt, bald werden Sie mit dieser Registeranzeige auf vertrautem Fuß stehen.

PC	IRQ	NV-BDJZC	AC	XR	YR	SP
E147	EA31	10110000	00	00	00	F8

Tab. 2: eine Registeranzeige

N	V	-	B	D	J	Z	C
Negativ-Flagge	Überlauf-Flagge	unbenutzt	Abbruch-Flagge	Dezimal-Flagge	Interrupt-Flagge	Zero-(Null-) Flagge	Carry-(Übertrag-) Flagge

Tab. 3: das Prozessor-Statusregister P – die Flaggen

Wie sieht ein Assemblerprogramm aus?
Das menschliche Gehirn hat dem des Computers Vieles voraus. Dazu gehört beispielsweise, dass ein Mensch allerlei Dinge gleichzeitig tun kann: gehen, sprechen, lächeln, Musik hören, Handbewegungen ausführen, womöglich dabei auch noch etwas kauen und so weiter.

Ein Computer ist dazu nicht im Stande. Er erledigt eine kleine Aufgabe nach der anderen. Weil er das so schnell macht, hat es für uns den Anschein, es geschähe alles gleichzeitig.

Das Maschinenprogramm ist eine Kette solch kleiner Aufgaben. Das erste Glied daraus, das wir kennen lernen wollen, ist der Befehl LDA.

Das bedeutet »LaDe den Akkumulator«. Alle Assembler-Befehlsworte bestehen aus drei Buchstaben, wie dieser hier auch. Wir haben in der ersten Folge schon gesagt, dass einem solchen Befehl eine 8-Bit-Kodezahl entspricht. Das ist hier $A9 oder binär 1010 1001 oder schließlich dezimal 169. Die Kodezahl muss in einem Speicherplatz stehen, zum Beispiel in $1500 (entspricht dez. 5376). Assemblerlistings sehen dann so aus:

```
1500 LDA
```

Hier tritt also die Speicherplatznummer mit einem nachfolgenden Befehl an Stelle der vom BASIC gewohnten Zeilennummer.

Hier fehlt noch etwas Entscheidendes: Was soll in den Akku geladen werden? Genauso wie es in BASIC Befehle gibt, die für sich alleine stehen können, wie »CLR« oder »LIST«, gibt es auch im Assembler solche Befehle. Weitaus häufiger aber sind hier Befehle, die ein Argument erfordern (in BASIC zum Beispiel »PEEK (100)«). Dabei ist 100 das Argument). In Assembler gibt es zwei Sorten von Argumenten: solche, die in einem Speicherplatz unterzubringen sind und andere, die zwei Bytes brauchen. Mit dem Befehlswort (hier also LDA) zusammen gezählt, existieren in Assembler also 1-Byte-Befehle, 2-Byte-Befehle und 3-Byte-Befehle.

Das Argument von LDA ist also das, was in den Akku soll. Laden wir also mal eine 1 in den Akku:

```
1500 LDA #$01
```

Wir haben jetzt einen 2-Byte-Befehl erzeugt. Was aber bedeuten # und $ dabei? $ ist leicht zu erklären. Die große Mehrzahl der Assembler nimmt bei Zahlenangaben Hexadezimalzahlen an. Bei einigen muss man das durch das $-Zeichen kennzeichnen. Manche Assembler lassen auch Binärzahlen, Dezimalzahlen und sogar ASCII-Zeichen als Argumente zu. Für jede Eingabeart steht dann vor dem Argument ein Zeichen, das die Art des Arguments angibt, zum Beispiel häufig »!« für Dezimalzahlen oder »%« für Binärzahlen.

Nun zum #-Zeichen. Es gibt viele Arten, den Akku zu laden. Direkt mit einer Zahl - wie hier -, aber zum Beispiel auch mit dem Inhalt eines anderen Speichers und so weiter. Man spricht von der so genannten Adressierung.

Es gibt eine ganze Menge davon, und jede wird auf eindeutige Weise gekennzeichnet. Wenn wir in unseren Akku eine Zahl laden, dann ist das die »unmittelbare« Adressierung, und die kennzeichnet man mit dem #-Zeichen.

Wenn in Speicherstelle $1500 die Kodezahl für LDA steht, dann muss die 1 in der Speicherstelle $1501 stehen, wie es sich für einen 2-Byte-Befehl gehört. Wenn Sie nun die Assemblerzeile eingegeben haben und <RETURN> drücken, dann taucht auf dem Bildschirm ... eine Fehlermeldung auf (bei vielen Assemblern). Wir müssen nämlich unser Software-Instrument vorher noch anweisen, jetzt zu assemblieren. Wie das geschieht, ist auch wieder von Assembler zu Assembler verschieden. Die meisten erwarten, dass man vor der Zeile noch ein A eingibt:

```
A 1500 LDA #$01
```

Wenn Sie jetzt <RETURN> drücken, zeigt der Bildschirm:

```
A 1500 LDA #$01                A 1502
```

und meistens einen blinkenden Cursor, der auf die nächste Eingabe wartet. $1502 ist die nächste freie Speicherstelle, und wenn beim Programmablauf der Programmzähler nach dem LDA #$01 auf $1502 deutet, dann erwartet er dort den nächsten Befehl. Wenn dort Unsinn steht, dann stürzt der Computer im Allgemeinen ab, je nachdem, welcher Kode dann hier zufällig enthalten ist. Wir haben ja 256 Möglichkeiten dafür: $00 bis $FF.

Im Gegensatz zu BASIC, wo man durch den Interpreter die Möglichkeit hat, Zeilennummern zu bauen, wie man will, muss hier das Progamm eine ununterbrochene Perlenschnur von Befehlen in Speicherstellen sein. Durch einige Befehle lässt sich dieses Prinzip allerdings durchbrechen.

Damit wir die Wirkung von Befehlen sehen können, greife ich auf einen Befehl vor, der ähnlich dem »STOP« in BASIC einen Programmabbruch bewirkt: BRK. Die genaue Funktion soll erst später erklärt werden - wir erhalten jedenfalls dann, wenn ein Maschinenprogramm auf einen BRK-Befehl läuft, die Registerinhalte angezeigt. Das ist in den meisten Assemblern eingebaut. Wir ergänzen jetzt:

```
A 1502 BRK
```

Damit erstmal genug. Steigen Sie aus dem Assembler aus und starten Sie das Programm. In den meisten Assemblern geht das mit G 1500 oder von BASIC aus mit »SYS 5376«.

Jetzt werden wieder die Register angezeigt. Der Programmzähler steht auf 1503, im Akku steht 01, alle Flaggen außer der Breakflagge sind null (die unbenutzte Flagge steht immer auf 1). Jetzt ändern wir das Argument:

```
A 1500 LDA #$00 A 1502 BRK
```

Wir starten wieder und sehen uns die Register an: Programmzähler 1503, Akku jetzt 00, aber bei den Flaggen hat sich etwas verändert: Die Zero-Flagge ist auf 1 gesetzt. Wir sehen also: Diese Flagge bleibt solange ungesetzt, solange nicht eine Null im Akku auftaucht, erst dann wird sie 1. Noch einmal ändern wir das Programm:

```
A 1500 LDA #$FF A 1502 BRK
```

Nach erneutem Start steht das Erwartete in den Registern, nur bei den Flaggen ist etwas Merkwürdiges passiert: Die Vorzeichenflagge steht auf 1. Das bedeutet, im Akku soll eine negative Zahl stehen! Nun wissen wir aber, dass $FF = dez. 255 ist. Dieses Rätsel wird uns noch eine Weile begleiten. Es sei hier nur bemerkt, dass kein Fehler vorliegt: Immer, wenn in einer Zahl das Bit 7 gleich 1 ist, wird die Vorzeichenflagge auf 1 gesetzt. Die Lösung des Rätsels werden wir bei den negativen Binärzahlen finden.

Wir schließen aus alledem: Der LDA-Befehl beeinflusst die Vorzeichen- und die Zeroflagge.

Der zweite Assemblerbefehl: STA

STA heißt »STore Accumulator«, also »lege Akkuinhalt ab«. Wie Sie sich denken können, muss auch hier ein Argument auftauchen: nämlich, wohin abgelegt werden soll. Wir legen unseren Akkuinhalt in die erste Bildschirmspeicherstelle (C 64: $0400, VC 20-Grundversion: $1E00, VC 20 mit Erweiterung: $1000). Unser Programm muss also so aussehen:

```
A 1500 LDA #$01
A 1502 STA $0400    oder die entsprechende Adresse
                    (siehe oben)
```

Mit dem STA-Befehl lernen wir eine neue Adressierungsart kennen: die »absolute« Adressierung. Sie ist daran zu erkennen, dass kein besonderes Merkmal verwendet wird. Die Adresse $0400 ist nicht in einem Byte darstellbar, sondern wird aufgeteilt auf zwei Bytes. Im Speicher steht jetzt:

```
LDA #
$ 01
STA
$ 00 (das ist das LSB)
$ 04 (das ist das MSB)
```

Hier liegt also ein 3-Byte-Befehl vor, und die nächste freie Speicherstelle ist $1505.

Vom BASIC her wissen Sie, dass 1 der Bildschirmkode für den Buchstaben A ist, und dass man jeder Bildschirmspeicherstelle auch eine Bildschirmfarbspeicherstelle zuordnet. Um ein eingeschriebenes Zeichen vom Hintergrund abzuheben, muss man dort eine Farbinformation eingeben. Der Start dieses Bildschirmspeichers liegt so:

C 64:	$D800
VC 20 (Grundversion):	$9400
VC 20 (erweiterte Version):	$9600

Der Farbe Schwarz entspricht die Kodezahl 0. Wir ergänzen unser Programm durch:

```
A 1505 LDA #$00
A 1507 STA $D800 (oder entsprechender Speicher)
```

Die nächste freie Adresse ist nun $150A. Unser Programm soll jetzt abgeschlossen sein. Damit der Computer aber beim Programmzählerstand $150A nicht Unsinn vorfindet, muss - ähnlich wie bei »END« in BASIC - das Programm auf irgendeine Weise beendet werden. Das kann durch BRK geschehen. Wir wollen aber den dritten Assembler-Befehl kennen lernen: RTS. Das heißt »ReTurn from Subroutine«, also »Rückkehr aus Unterprogramm«. In unserem Fall bewirkt das eine Rückkehr zum BASIC. Wie Sie sehen, ist das ein 1-Byte-Befehl, also ohne Argument.

Auch hier spricht man von einer Adressierungsart, nämlich der »impliziten« Adressierung. Man erkennt sie am Fehlen des Arguments. Die Adresse ist implizit, das heißt im Befehl selbst enthalten. Dies ist nämlich ein Befehl, der immer an den Programmzähler gerichtet ist.

Der Computer holt sich vom Stapelspeicher die dort zuoberst liegende Adresse, das ist die, bei der der Computer in ein Unterprogramm gesprungen ist oder aber die, bei der der Computer BASIC verlassen hat. Wir ergänzen also noch:

```
A 150A RTS
```

und starten das Programm, zum Beispiel von BASIC aus mit »SYS 5376«.

Natürlich taucht dann in der linken oberen Ecke des Bildschirms ein schwarzes A auf. Hier noch der BASIC-Lader:

```
10 FOR I = 5376 TO 5386 : READ A : POKE
I,A : NEXT I : END
20 DATA 169, 1, 141, 0, 4*, 169, 0, 141,
0, 216*, 96
```

Die mit * markierten Zahlen müssen für den VC 20 verändert werden: Grundversion: 30 und 148, erweiterte Version: 16 und 150.

Eine Kombination von LDA mit STA ist vergleichbar mit dem POKE-Befehl in BASIC. Man kann in Assembler nicht direkt eine Zahl in einen Speicher einschreiben, sondern muss den Umweg über den Akku machen.

Außer dem Akku eignen sich dazu aber auch das X-Register und das Y-Register. Hierfür gibt es die Befehle LDX (lade X-Register), STX (lege X-Register-Inhalt ab), LDY (lade Y-Register) und schließlich STY (lege Y-Register-Inhalt ab).

Sie können das übungshalber an unserem kleinen Programm ausprobieren.

An dem folgenden Programm sehen Sie noch eine Eigenart der drei Register (Akku, X-Register, Y-Register):

```
A 1500 LDA #$01
A 1502 LDX #$00
A 1504 LDY #$02
A 1506 STA $0400
A 1509 STX $D800
A 150C STY $0401
A 150F STX $D801
A 1512 STA $0402
A 1515 STX $D802
A 1518 RTS
```

Für den VC 20 werden die entsprechenden Speicherstellen für Bildschirmspeicher und Bildschirmfarbspeicher eingesetzt. Dieses Programm druckt - wie erwartet - »ABA« in die linke obere Ecke des Bildschirms. Dabei ist das X-Register dreimal ausgelesen worden und der Akku zweimal.

Sie sehen also, dass die Registerinhalte durch die STA-, STX- und STY-Befehle nicht verändert werden.

Wir wollen noch etwas ausprobieren. Bisher haben wir den LDA-Befehl nur mit der »unmittelbaren« Adressierung kennen gelernt. LDA, LDX und LDY können jedoch auch »absolut« adressiert werden:

```
A 1518 LDA $D800
```

Damit laden wir den Inhalt der Speicherstelle $D800 (beim VC 20 die anderen Adressen des Bildschirmfarbspeichers) in den Akku. Der Inhalt ist seit $1509 eine Null. Jetzt weiter:

```
A 151B STA $0403
A 151E STX $D803
A 1521 RTS
```

Das müsste beim Ablauf des Programms noch einen Klammeraffen (@mit Bildschirmkode 0) an die vierte Stelle platzieren, was Sie durch »SYS 5376« leicht nachprüfen können. Sie sehen, dass man mit diesen sieben Befehlen schon eine Menge anfangen kann.

Wir kommen noch einmal zur Adressierung. Ich hatte Ihnen gesagt, dass LDA #$01 ein 2-Byte-Befehl mit unmittelbarer Adressierung ist (ein Byte für LDA und eines für 01), LDA $D800 ist ein 3-Byte-Befehl (ein Byte für LDA, je eines für das LSB und das MSB von $D800) mit absoluter Adressierung. Da werden Sie sich doch sicher schon gefragt haben, wo die Adressierung bleibt.

Wenn aber kein Byte für die Adressenmarkierung (zum Beispiel #) reserviert ist, muss die Kennzeichnung irgendwie anders sein.

Wenn Sie einen Disassembler zur Verfügung haben, dann sehen Sie sich damit unser Programm an. Fast jeder Disassembler gibt neben dem Assemblertext auch Byte für Byte in Hexadezimalzahlen die Kodes an. Wenn Sie nun die beiden Befehle LDA #$01 und LDA $D800 von den Kodes her untersuchen, sehen Sie Folgendes:

```
1500 A9 01        LDA #$01
1518 AD 00 D8     LDA $D800
```

Offensichtlich gehört jeweils das erste angezeigte Byte zu LDA. Sie sind aber verschieden!

Wir sehen daraus, dass die Kodezahl für einen Befehl gleich zwei Informationen enthält: Das Befehlswort selbst (LDA) und die Adressierungsart.

Genauso, wie man LDA sowohl unmittelbar als auch absolut ausführen kann, ist das auch mit LDX und LDY möglich. Bei den Befehlen STA, STX und STY ist eine unmittelbare Adressierung sinnlos. Für RTS kennt man nur eine implizite Adressierung. Wir fassen das alles zusammen in Tabelle 4.

In den letzten Spalten von Tabelle 4 ist noch angegeben, inwieweit durch diese Befehle das Prozessorstatusregister beeinflusst wird, so wie wir es für den Befehl LDA schon ausprobiert haben. In der vorletzten Spalte sehen Sie, wie lange die Ausführung eines

Tab. 4: die ersten sieben Befehle

Befehlswort	Adressierung	Byte-Anzahl	Kode		Dauer in Taktzyklen	Beeinflussung von Flaggen
			HEX	DEZ		
LDA	unmittelbar	2	A9	169	2	N, Z
LDA	absolut	3	AD	173	4	N, Z
LDX	unmittelbar	2	A2	162	2	N, Z
LDX	absolut	3	AE	174	4	N, Z
LDY	unmittelbar	2	A0	160	2	N, Z
LDY	absolut	3	AC	172	4	N, Z
STA	absolut	3	8D	141	4	keine
STX	absolut	3	8E	142	4	keine
STY	absolut	3	8C	140	4	keine
RTS	implizit	1	60	96	6	keine

Befehls dauert. Wenn sie für einen Taktzyklus etwa eine Mikrosekunde rechnen, dann müssten Sie jetzt ausrechnen können, wie lange unser letztes Programm zur Bearbeitung braucht: 48 Mikrosekunden. Ein vergleichbares BASIC-Programm braucht dazu etwa hundertmal so lange: zirka 0,05 Sekunden.

Ein bisschen von Assembler-Alchemie verstehen Sie jetzt schon mit diesen sieben Befehlen. Wir wollen uns nun die Zahlen ansehen, die hier Verwendung finden: Das Binärsystem und das Hexadezimalsystem.

Die einzigen Ziffern, die unser Computer kennt, sind 0 und 1. Sie stehen für »Strom an« oder »Strom aus« oder für »keine magnetische Erregung« bzw. »magnetische Erregung«.

Deswegen ist es für uns als angehende Assembler-Alchemisten von großer Bedeutung - wir arbeiten ja ganz eng an der Hardware -, dieses binäre Zahlensystem handhaben zu können.

Das Hexadezimalsystem kennt der Computer eigentlich gar nicht. Wir verwenden es, weil es in einem besonders engen Zusammenhang mit Binärzahlen und dem Aufbau unseres Computers steht: Die größte einstellige Hex-Zahl ist $F, das entspricht genau 1111 im Binärsystem, also dem maximalen Füllungsgrad eines halben Bytes, das »Nibble« genannt wird. Ein ganzes Byte kann maximal $FF enthalten (binär 1111 1111) und der gesamte Speicheradressenbereich unseres Computers geht bis $FFFF (dezimal 65535).

Eine einstellige Hex-Zahl passt also in ein Nibble, eine zweistellige in ein Byte und eine dreistellige oder vierstellige in zwei Bytes, weshalb man solche Hex-Adressen auch recht leicht in das LSB und das MSB aufteilen kann:

```
$D8 00
```

wobei D8 das MSB und 00 das LSB ist. Rechnen werden wir mit Hexadezimalzahlen nicht. Wir benutzen dazu das Dezimalsystem oder - wenn es sich um computerinterne Vorgänge handelt - das Binärsystem.

Das Rechnen mit Binärzahlen funktioniert genauso wie das mit Dezimalzahlen. Es gilt also:

```
0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 10
```

wobei binär 10 gleich dezimal 2 ist. Als Beispiel können wir 2 + 1 = 3 im Binärsystem rechnen:

```
  10 (entspricht dez. 2)
+ 01 (entspricht dez. 1)
  11 (entspricht dez. 3)
```

Die Addition erfolgt also spaltenweise wie beim gewohnten dezimalen Addieren. Auch mit dem Übertrag läuft es wie im Dezimalen. Beispiel 2 + 2 = 4:

```
  10 (entspricht dez. 2)
+ 10 (entspricht dez. 2)
 100 (enstpricht dez. 4)
```

In der zweiten Spalte wurde nach der Regel verfahren: 1 + 1 = 10. Rechnen wir nun noch 3 + 3 = 6:

```
  11 (entspricht dez. 3)
+ 11 (entspricht dez. 3)
 110 (entspricht dez. 6)
```

In der ersten Spalte wurde gerechnet 1 + 1 = 10, wobei nach dem alten Motto: 0 hin, 1 im Sinn die 0 unter den Strich gesetzt wurde. In der zweiten Spalte wird dann so verfahren: 1 + 1 + 1 (das ist die 1, die wir »im Sinn« hatten) = 11. Ich meine, dass Sie ohne Probleme die folgenden Übungsaufgaben lösen und dann jeweils dezimal das Ergebnis nachprüfen können: 10 + 5, 7 + 1, 16 + 16, 240 + 16, 62 + 65

In der nächsten Folge werden wir eine Anzahl neuer Assembler-Befehle kennen lernen und erfahren, wie der Computer Zahlen voneinander abzieht. Bei der Gelegenheit lernen Sie dann noch einige Flaggen kennen, und wir werden das Rätsel, warum bei `LDA # $FF` eine negative Zahl angezeigt wird, lösen. Und vor allem: Sie erhalten einen komfortablen Assembler, Disassembler und einen Monitor.

Assembler ist keine Alchemie
Folge 3

Erschienen in: 64'er 11/1984
Verantwortlicher Redakteur: Georg Klinge
Autor: Heimo Ponnath

In den ersten beiden Folgen unseres Assembler-Kurses mussten Sie noch mit BASIC-Ladern arbeiten. Jetzt steht Ihnen ein leistungsfähiger Monitor zur Verfügung, der SMON. Somit können Sie alle Beispiele direkt eingeben und ausprobieren.

In der letzten Folge haben wir die ersten Assembler-Befehle kennen gelernt und wissen, wie man sie benutzt und was sich im Computer dabei tut. Die Zahlen der Assembler-Alchemisten haben uns einige Geheimnisse enthüllt, obwohl sie für die Zweifingerlinge und die Sechzehnfingerlinge gedacht sind. Die Binärzahlen können wir schon zusammenzählen.

Heute werden Sie eine Reihe weiterer Assembler-Befehle kennen lernen und noch ein weiteres Zahlensystem. Wir ergründen das Geheimnis der negativen Zahlen und machen uns die Funktion der Flaggen zu Nutze.

Wir haben nun auch einen sehr brauchbaren Assembler für den C 64: den SMON. **Sie können ihn als D64-Diskettenimage unter www.skriptorium-vd.de/01/smon.d64 herunterladen. Eine Anleitung zum Programm finden Sie weiter hinten im Buch.** Künftig wird in dieser Serie die SMON-Syntax verwendet und kein BASIC-Lader mehr angegeben. Außerdem hat in Ausgabe 9 die Serie »Der gläserne VC 20« begonnen, so dass sich der Schwerpunkt hier mehr auf den C 64 verlagert. Das sollte aber die VC 20-Fans nicht davon abhalten, diesen Kurs weiter zu verfolgen, denn bis auf gelegentliche Adressänderungen ist fast alles für sie verwendbar.

Eine Zauberformel der Assembler-Alchemisten: INX, INY, INC, DEX, DEY, DEC

Wir wissen ja schon, dass man diese »Zauberformeln« entzaubern kann. `INX` heißt einfach »INcrement X-Register«, also Inhalt des X-Registers um 1 erhöhen.

Es wird Ihnen sicher einleuchten, dass `INY` dasselbe mit dem Y-Register tut. Etwas weniger deutlich ist das bei `INC`. Das bedeutet »INCrement memory«, also zähle zum Inhalt einer Speicherstelle eins dazu. `INX` und `INY` enthalten alles, was dem Computer zu sagen ist, sind also offensichtlich 1-Byte-Befehle mit der in der letzten Folge schon kennen gelernten impliziten Adressierung. Bei `INC` muss dem Computer noch gesagt werden, welche Speicherstelle er um 1 erhöhen soll. Es gehört also noch eine Adresse dazu. Das lässt diesen Befehl im Allgemeinen zu einem 3-Byte-Befehl werden.

Befehle zum Zählen

Das umgekehrte leisten die Befehle `DEX`, `DEY` und `DEC`. Sie bedeuten nämlich »DEcrement X-Register«, also »zähle das X-Register um eins herunter« beziehungsweise das Y-Register oder - bei `DEC` - die angegebene Speicherstelle. Für die Adressierungsart und die Anzahl Bytes pro Befehl gilt hier das gleiche wie für die `INX` ...-Befehle. Sehen wir uns das an einem kleinen Beispiel an (Bitte lesen Sie sich dazu die Bedienungshinweise zum SMON durch):

```
1500 LDA #00
1502 LDX #01
1504 STA D800
1507 STX 0400
150A INX
150B STA D801
150E STX 0401
1511 DEX
1512 STA D802
1515 STX 0402
1518 BRK
```

Wenn Sie das kleine Programm mit G 1500 starten, sollten Sie in der linken oberen Ecke des Bildschirms »ABA« in schwarzer Schrift stehen haben. Was ist geschehen?

Wir haben den Inhalt des Akkus (= 0, also Farbkode für schwarz) in das Bildschirm-Farbregister geschrieben (#D800), dann den Inhalt des X-Registers (1 = POKE-Kode für den Buchstaben A) in die erste Bildschirm-Speicherzelle (# 0400).

Anschließend wurde das X-Register um 1 erhöht (2 = POKE-Kode für den Buchstaben B) und dieser Inhalt in die zweite Bildschirmzelle geschrieben. Außerdem musste natürlich auch dieser Bildschirm-Farbspeicherplatz mit dem Farbkode 0 belegt werden.

Durch DEX wurde das X-Register wieder heruntergezählt, somit wieder ein A erzeugt und in die dritte Bildschirmstelle gedruckt.

Sie haben sicher schon bemerkt, dass man auf diese Weise Abläufe mitzählen kann. Soll zum Beispiel ein Vorgang 20-mal wiederholt werden, dann packt man ins X-Register (oder ins Y-Register oder in eine andere Speicherstelle) den Anfangswert 0, lässt den Computer eine Arbeit ausführen, erhöht das entsprechende Register oder die Speicherzelle um 1 mit INX, INY oder INC, prüft dann, ob dieser Inhalt schon 20 geworden ist und so weiter. Wie man diese Prüfung vornimmt, dazu kommen wir erst später bei den »Branch«-Befehlen. Das ist also ähnlich wie im BASIC bei den FOR ... NEXT-Schleifen: Dort wird eine Variable als Zähler verwendet, hier ein Register (oder eine Speicherstelle). Ebenso wie im BASIC bei diesen Schleifen kann man auch hier rückwärts zählen: mit DEX, DEY oder DEC. Das hat oft gewisse Vorzüge, was uns aber noch nicht kümmern soll.

Wenn wir diese Befehle als Zähler verwenden, sollten wir im Auge behalten, dass eine Speicherstelle (auch ein X- oder Y-Register) Zahlen nur von 0 bis 255 enthalten kann. Die höchste 8-Bit-Zahl ist ja

```
          1111 1111 (bin.) / dez. 255
+                    1            1
= (1) 0000 0000                  0!
```

Wenn wir also über 255 hinaus zählen, ergibt sich wieder 0 und so weiter, weil ein Überlauf stattgefunden hat. Das neunte Bit passt nicht mehr in das Byte hinein.

Um noch mal genau sehen zu können, was unser Computer da tut, probieren Sie einmal aus:

```
1500 LDA #01
1502 BRK
```

Das soll uns die Register zunächst mal im Ausganszustand zeigen. Nach G 1500 werden sie angezeigt:

```
AC XR YR N V - BD I ZC
01 00 00 0 0 1 10 0 00
```

Im Akku steht jetzt die dort eingeladene 1. Nun wollen wir das X-Register mit 255 (also $FF) laden. Dazu ändern wir das Programm:

```
1502 LDX #FF
1504 BRK
```

Nach erneutem G 1500 zeigen die Register:

```
AC XR YR N V - BD I ZC
01 FF 00 1 0 1 10 0 00
```

Im X-Register steht nun die Zahl $FF. Bei den Flaggen hat sich die N-Flagge (die negative

Zahlen anzeigen soll) auf 1 geschaltet! Nun wollen wir das X-Register über 255 hinaus zählen. Wir verändern das Programm noch mal:

```
1504 INX
1505 BRK
```

Der Start mit G 1500 liefert uns die folgende Registeranzeige:

```
AC XR YR N V - BD I ZC
01 00 00 0 0 1 10 0 10
```

Wie erwartet, ist der Überlauf des X-Registers eingetreten: Es ist jetzt Null. Die N-Flagge hat ihren gewohnten Wert 0 wieder angenommen und die Z-Flagge, die uns anzeigt, ob die letzte Operation eine Null erzeugt hat, ist jetzt gesetzt. Bei weiterem Hochzählen verschwindet die Z-Flagge wieder:

```
1505 INX
1506 BRK
```

G 1500 liefert den Registerinhalt:

```
AC XR YR N V - BD I ZC
01 01 00 0 0 1 10 0 00
```

Das Gleiche passiert bei Verwendung des Y-Registers als Zähler, wie Sie leicht durch Austauschen aller auf X bezogenen Befehle feststellen können.

Sehr nett ist es, diesen Befehlsablauf einmal für den INC-Befehl auf die Speicherstelle $0400 (Bildschirmspeicher links oben) bezogen ablaufen zu lassen. Wenn man darauf achtet, dass kein Hochscrollen des Bildschirms eintritt, kann man das Ergebnis außer in den Registern auch noch als Zeichen auf dem Bildschirm verfolgen. Der Beginn der Befehlssequenz ist dann sinnvollerweise:

```
1500 LDA #FF
1502 STA 0400
1505 BRK
```

Im Folgenden setzt man dann an Stelle von INX immer INC 0400 ein.

Was passiert beim Herunterzählen unter Null? Sie können das mit der gezeigten Befehlskette leicht verfolgen, indem Sie immer statt INX den Befehl DEX setzen und die Register nicht mit $FF, sondern mit 01 laden.

Es zeigt sich, dass beim Herunterzählen nach der Null wieder 255 (= $FF) im Register zu finden ist. Die Reaktion der N- und der Z-Flagge auf den jeweiligen Registerinhalt ist die Gleiche wie beim Hochzählen.

Es ist nun deutlich, dass diese sechs Befehle die N-Flagge und die Z-Flagge beeinflussen können. Diese Tatsache wird später noch eine große Rolle spielen, wenn es um die bereits erwähnte Schleifenkontrolle geht.

Noch ein alchemistischer Zahlentrick

Die Assembler-Alchemisten haben noch viel mehr Arten der Zahlen- und Zeichendarstellung auf Lager. Eine davon ist die Kodierung als »BCD-Zahlen«.

BCD kommt vom englischen »Binary Coded Dezimal«, was bedeutet: binär kodierte Dezimalzahlen.

Zwischendurch möchte ich noch eine Bemerkung loswerden, die Sie als Trost auffassen sollen: Auch wenn wir später andere Zahlendarstellungen kennen lernen werden, es wird nicht so schwierig! Sogar so komplette Idioten wie Computer verstehen das, obwohl man ihnen alles haarklein vorkauen muss.

Wenden wir uns nun wieder den lächerlich einfachen BCD-Zahlen zu. Alle Zahlen von 0 bis 9 lassen sich binär mit nur 4 Bits ausdrücken:

Dezimal	Binär	Dezimal	Binär
0	0000	5	0101
1	0001	6	0110
2	0010	7	0111
3	0011	8	1000
4	0100	9	1001

Tab. 5: Vergleich von Dezimal- und Binärzahlen

Die weiteren Werte 1010 bis 1111 werden in der BCD-Kodierung nicht benutzt. Liegt nun eine Dezimalzahl (zum Beispiel 12) vor, dann wird jede Stelle dieser Zahl (also die 1 und die 2) getrennt binär kodiert.

In unserem Beispiel mit der 12 wäre das dann 0001 für die 1 und 0010 für die 2. Somit ist die 12 im BCD-Kode 0001 0010. Jede Ziffer erhält so ihr Nibble. Eine Zahl im BCD-Format hat deswegen keine feste Anzahl von Bytes, sondern die Byte-Zahl hängt von der Anzahl der Stellen ab. Die Zahl 1984 beispielsweise braucht 2 Bytes: 0001 1001 1000 0100.

Schwierig gestaltet sich das Rechnen mit diesen Zahlen wegen der sechs unbenutzten Kodes. Aber auch da habe ich einen Trost für Sie: Wir werden damit nicht rechnen. Wozu das Ganze dann, werden Sie sich fragen.

Der Grund für das alles ist, dass BCD-Zahlen im Gegensatz zu den Zahlen mit festem Format (die sonst verwendet werden) so eingegeben und verarbeitet werden können, wie sie vorliegen. Das ist im kaufmännischen Bereich manchmal notwendig, wo eben 1000 x 0,1 Pfennige 1 Mark ergeben und Fehler unzulässig sind. Sollten Sie also vor dem Problem stehen, mit BCD-Zahlen rechnen zu müssen, grämen Sie sich nicht: Unser Prozessor kennt den Dezimalmodus. Er ist dann eingeschaltet, wenn die Dezimal-Flagge auf 1 gesetzt ist.

Damit sollen Sie dann auch noch gleich zwei neue Befehle kennen lernen: SED und CLD. Der Erstere hat nichts mit Parteien zu tun, sondern ist die Abkürzung für »SEt Dezimal flag«, also setze die Dezimalflagge. So schalten Sie den Dezimal-Modus ein. Wie Sie sicher schon messerscharf geschlossen haben, heißt CLD »CLear Dezimal flag«, also setze die Dezimalflagge auf Null, wodurch dieser Modus wieder auszuschalten ist.

Wichtig! Wenn Sie argwöhnen, dass in einem Programm irgendwann mal die Dezimal-Flagge gesetzt sein könnte, dann gehen Sie auf Nummer sicher und schieben vor eine Rechenoperation, die nicht im Dezimalmodus laufen soll, ein CLD.

Beide Befehle sind 1-Byte-Befehle mit implizierter Adressierung. Sie beeinflussen lediglich die Dezimalflagge.

Das Geheimnis der negativen Binärzahlen

Wie schon mal betont: Der Computer ist strohdumm. Er kann nicht einmal auf normale Weise voneinander abziehen! Deswegen geht er den komplizierten Weg: Er addiert eine negative Zahl.

Nur: Wie sehen negative Binärzahlen aus? Wir werden diese Frage in drei Etappen beantworten.

a) Man könnte eine Flagge setzen, die bei negativen Zahlen 1 ist und bei positiven Zahlen 0. Bei einigen Fließkommazahlen wird das auch so gemacht.

Hier aber setzt man die Flagge direkt in die Zahl ein: Bit 7 jeder Zahl ist jetzt ein Vorzeichenmerkmal. Wenn dieses Bit 0 ist, handelt es sich um eine positive, wenn es 1 ist, um eine negative Zahl. Auf diese Weise ist also +1 wie bisher 0000 0001, wohingegen -1 jetzt 1000 0001 hieße.

Damit wird allerdings der Zahlenbereich, der durch ein Byte auszudrücken ist, verschoben. 255 = binär 1111 1111 kann so nicht mehr verwendet werden.

Die größte Zahl, die jetzt ausgedrückt werden kann, ist 0111 1111 = dezimal 127. Die kleinste Zahl ist dann 1111 1111 = -127. Probieren wir mal aus, wie sich damit rechnen lässt:

```
+10    0000 1010
- 6    1000 0110
=      1001 0000   (= -16)
```

was offensichtlich falsch ist, denn nach Adam Riese sollte +4 herauskommen. So kann man also nicht rechnen!

Man nennt diese Art der Zahlendarstellung übrigens »Signed Binary«-Format, also auf Deutsch: markierte Binärzahlen.

b) Der nächste Schritt ist das so genannte Einerkomplement. Dabei tritt für die positiven Zahlen keine Änderung ein. Die negativen entstehen aus den positiven durch Komplementbildung, das heißt jedes Bit der positiven Zahl wird in sein Gegenteil verkehrt, wie es das folgende Beispiel zeigen soll:

```
0000 1100 = +12
```

dann ist das Einerkomplement:

```
1111 0011 = -12
```

Komplement ist nicht kompliziert

Interessanterweise taucht hier auch wieder das Merkmal der »Signed Binary«-Zahlen auf: die 1 in Bit 7 bei negativen Zahlen. Beschränkt man sich auf den Zahlenbereich, der für die »Signed Binary«-Zahlen gültig war, dann hätten wir jetzt beide Darstellungsweisen miteinander vereint. Nun müssen wir natürlich noch feststellen, ob man so auch rechnen kann.

```
+8      0000 1000
-6      1111 1001  (als Einerkomplement)
= (1)   0000 0001
```

was 1 mit einem Übertrag ergäbe - jedenfalls nicht 2, wie's sich gehört. Also ist auch die Einerkomplementdarstellung noch nicht das Gelbe vom Ei.

c) Ich will Sie nicht länger auf die Folter spannen: Wenn man zum Einerkomplement einer Zahl noch 1 dazuzählt, erhält man das Zweierkomplement. Und genau so werden negative Zahlen in unserem Computer gehandhabt. Die positiven Zahlen bleiben unverändert. Von den negativen bildet man das Zweierkomplement wie zum Beispiel hier mit der Zahl -12:

```
  12    0000 1100  (binär)
(-12)   1111 0011  (Einerkomplement)
 + 1    0000 0001  addieren
 -12    1111 0100  (Zweierkomplement)
```

Jetzt wollen wir auch diese Zahlenart ausgiebig testen. Wir rechnen noch mal 8 - 6:

```
+ 8     0000 1000
- 6     1111 1010  (Zweierkomplement)
=  (1)  0000 0010  also 2 mit einem
                   Übertrag, der igno-
                   riert wird.
```

Das Ergebnis ist richtig. Wenn bei einer solchen Rechnung eine negative Zahl herauskommt, ist sie nicht leicht zu erkennen. In solchen Fällen kehrt man das Vorzeichen um, indem man das Zweierkomplement berechnet. Das machen wir mal am Beispiel 5 - 6:

```
+ 5     0000 0101
- 6     1111 1010  (Zweierkomplement)
=       1111 1111  (-1 als Zweierkomp-
                   lement)
```

Zur Kontrolle nun die Vorzeichenumkehr durch Umrechnen ins Zweierkomplement:

```
        0000 0000  (Einerkomplement von -1)
+1      0000 0001
=       0000 0001  (also wie erwartet +1)
```

Auf diese Weise rechnet unser Computer mit negativen Zahlen. Negative ganze Zahlen speichert er im Zweierkomplement-Format. Auch wenn wir nun etwas vorgreifen müssen, wollen wir uns das ansehen. Dazu schalten Sie am besten erst einmal den Computer aus und laden dann den SMON beziehungsweise Ihren Assembler. Dann bauen wir ein kleines BASIC-Programm:

```
10 A% = -12
20 END
```

Wie Variablen im Speicher stehen

Noch nicht »RUN« eingeben! Zuerst schalten Sie den Maschinensprachemonitor ein, und wir sehen uns das Programm so an, wie es im Speicher steht. Der BASIC-Speicher des C 64 beginnt im Normalfall bei $0800.

Wir geben also den Monitorbefehl `M 0800` ein. Uns genügen schon die Speicherplätze bis $081C. Nun sehen wir das nackte BASIC-Programm im Speicher, so wie es uns C. Sauer in seinem Artikel »Der gläserne VC 20, Teil 1« in der 64'er 9/1984, Seite 156 beschrieben hat.

In Abbildung 5 ist unser Speicherinhalt kommentiert zu sehen. Das Programm endet im Speicherplatz $0813. Das Kennzeichen für Programmende sind zwei aufeinander folgende Bytes mit dem Wert Null.

Dahinter werden die Variablen abgelegt, sobald das Programm gestartet wird. Wir steigen aus dem Monitor durch Eingabe von »X« aus und starten das Programm mit »RUN«. Jetzt sehen wir nochmal in den Speicher. Bis $0813 hat sich nichts verändert. Danach aber ist jetzt in 7 Bytes die Variable A% abgelegt. Das zeigt Abbildung 6.

Zunächst einmal die Bytes $0814 und $0815: Hier werden der Variablenname und der -typ angegeben. Der Typ ist aus den Bits 7 zu erkennen. Sind beide (wie hier) gleich 1, dann handelt es sich um eine Integervariable (also eine ganze Zahl). Lässt man die Kennbits außer Acht, zeigt sich, dass in $0814 der Kode für den Buchstaben A steht und $0815 nur den Wert 0 enthält.

Abb. 5: Der Monitor zeigt das nackte Programm im Speicher.

0800	00	0C	08	0A	00	41	25	B2
		080C Koppeladresse		000A Zeilennr. 10		A	%	= Token
0808	AB	31	32	00	12	08	14	00
	– Token	1	2	Zeilen-ende	0812 Koppeladresse		0014 Zeilennr. 20	
0810	80	00	00	00	FF	FF	FF	FF
	END Token	Zeilen-ende	Programmende		leerer Speicher			

Nun zum Rest: Der C 64 legt Integers in nur 2 Bytes ab - die restlichen 3 Bytes $0818 bis $081A bleiben ungenutzt. Das ist auch dann der Fall, wenn danach noch weitere Variablen kommen. Es bringt also keine Speicherersparnis (VC 20-Benutzer aufgepasst!), wenn man mit Ganzzahlvariablen arbeitet!

In $0817 steht $F4, welches binär ausgedrückt `1111 0100` ist. Das kennen wir noch von weiter oben als die - 12 im Zweierkomplement-Format.

Woher kommt $FF in Speicherzelle $0816? Wie gesagt, die Integers werden in 2 Bytes gespeichert, und wenn wir -12 in 16 Bits ausdrücken, dann sieht das so aus:

```
+12   0000 0000 0000 1100
      1111 1111 1111 0011  (Einerkom-
                            plement)
+ 1   0000 0000 0000 0001
=-12  1111 1111 1111 0100
      MSB             LSB
    = $FF            =$F4
```

($FF und $F4 als 16-Bit-Zweierkomplement) Die größte positive ganze Zahl, die man in 2 Bytes ausdrücken kann, ist 32767, was binär `0111 1111 1111 1111` ergibt. Die kleinste ist `1000 0000 0000 0000`, also -32768.

Das ist der Grund dafür, dass der C 64 Integers größer als 32767 oder kleiner als -32767 dankend mit »ILLEGAL QUANTITY ERROR« ablehnt, wenn sie als Argument verwendet werden. (Die Zahl -32768 kann als Ergebnis logischer Operationen durchaus auftauchen.)

Damit will ich Sie für diesmal von den Zahlenspielereien erlösen.

In der nächsten Folge müssen wir darauf noch mal zurückkommen. Sie können die Art des Abziehens von Zahlen durch Addieren des Zweierkomplementes bis zum nächsten Mal an weiteren Beispielen üben. Wenn

Sie das mit 16-Bit-Zahlen tun, werden Sie bald feststellen, dass noch nicht alles so funktioniert, wie es sollte ...

Wir können jetzt übrigens auch das Rätsel lösen, weshalb bei positiven Zahlen (zum Beispiel LDA #FF) die Negativ-Flagge auf 1 geht: Die Flagge wird immer dann gezückt, wenn eine Zahl auftritt, die in Bit 7 eine 1 aufweist. Ganz einfach, gell?

Ein wirkungsvolles Zweiglein: BNE

Vermutlich raucht Ihnen nach soviel Zahlensalat der Kopf. Deshalb sollen Sie zur Entspannung noch einen neuen Assembler-Befehl kennen lernen und auch gleich ein nützliches Programmbeispiel dazu.

BNE heißt »Branch if Not Equal zero«, was man übersetzen kann mit »verzweige, wenn ungleich Null«. Genauer gesagt: Es wird dann verzweigt – also zu einer angegebenen Adresse gesprungen –, wenn die Z-Flagge (die haben wir bei den INX- / DEX ...-Befehlen genauer untersucht) nicht gesetzt ist, also 0 zeigt.

Sehen wir uns das mal an der folgenden Verzögerungsschleife an. Abbildung 7 zeigt das dazugehörige Flussdiagramm. Das Prögrämmchen dazu:

```
1500 LDX #FF
1502 LDY #FF
1504 DEY
1505 BNE 1504
1507 DEX
1508 BNE 1502
150A BRK
```

Zunächst werden das X- und das Y-Register als Zähler initialisiert, also mit einem Ausgangswert geladen.

Mit dem vorhin behandelten Befehl DEY wird dann das Y-Register um 1 heruntergezählt, was jetzt $FE ergibt. Für die Nullflagge (Z) bedeutet das den Inhalt 0, denn es liegt kein Grund vor, sie zu setzen (also dort eine 1 anzuzeigen), weil noch keine Null aufgetreten ist.

Bei der nachfolgenden Prüfung durch BNE wird also eine Verzweigung nach 1504 das Ergebnis sein, worauf das Y-Register weiter verringert und dann die Z-Flagge erneut geprüft wird und so weiter. Das geht so lange, bis nun wirklich endlich die Null im Y-Register erreicht ist.

In diesem Fall zählt DEX nun das X-Register herunter, und der nächste BNE-Befehl führt zum Sprung nach 1502, wo das Y-Register wieder auf $FF gesetzt wird. Auf diese Weise wird die äußere Schleife 255-mal und die innere 65025-mal durchlaufen.

Kein Widerspruch: Assembler-Programme langsamer machen

Sie haben beim Eingeben des Programms vermutlich etwas gestutzt, als der Assembler nach dem BNE 1504 als nächste Adresse statt dem erwarteten 1508 eine 1507 ausgegeben hat. Der Befehl sieht zwar wie ein 3-Byte-Befehl aus, ist aber nur ein 2-Byte-Befehl! Das liegt an der speziellen Art der Adressierung von solchen Branch-Anwei-

Abb. 6: So werden Integer-Variablen aus BASIC-Programmen vom C 64 im Speicher eingerichtet.

Speicherstelle	0814	0815	0816	0817	0818 – 081A
Byte	1	2	3	4	5 – 7
Inhalt	C1 1100 0001 Kennbits 7 für Integer 0100 0001 ≙ 65 Kode für A	80 1000 0000 0000 0000	FF 1111 1111 MSB von –12	F4 1111 0100 LSB	00 00 00 unbenutzt bei Integerzahlen
			Variablenname und -typ Variablenwert		

Tab. 6: die 11 neuen Befehle

Befehlswort	Adressierung	Byte-Anzahl	Kode		Dauer in Taktzyklen	Beeinflussung von Flaggen
			HEX	DEZ		
INX	implizit	1	E8	232	2	N, Z
INY	implizit	1	C8	200	2	N, Z
INC	absolut	3	EE	238	6	N, Z
DEX	implizit	1	CA	202	2	N, Z
DEY	implizit	1	88	136	2	N, Z
DEC	absolut	3	CE	206	6	N, Z
SED	implizit	1	F8	248	2	1 bis D
CLD	implizit	1	D8	216	2	0 bis D
BNE	relativ	2	D0	208	2	-
					+ 1 bei Verzweigung	
					+ 2 bei Überschreiten einer Seitengrenze	

sungen: der so genannten relativen Adressierung, die wir aber erst später mit den anderen Branch-Befehlen behandeln werden.

Wenn Sie das Programm mit G 1500 starten, werden Sie - obwohl alles in Maschinensprache schnell läuft - eine merkliche Verzögerung feststellen, bevor die Registeranzeige auftaucht. Noch längere Verzögerungen lassen sich ohne Weiteres erreichen, indem man mehrere Schleifen ineinander schachtelt. Dabei verwendet man dann den DEC-Befehl.

In Tabelle 5 sind auch die Zyklen angegeben, welche die heute neu gelernten Befehle zur Abarbeitung benötigen. Mit solchen Angaben lassen sich recht genau definierte Zeiten einstellen, in denen der Computer nichts anderes tut als durch das Programm zu flitzen.

Wozu das dient, braucht wohl kaum noch gesagt werden: Wenn Sie zum Beispiel einen Text auf dem Bildschirm lesen wollen, bevor das Programm weiterläuft oder wenn Sie mit Peripherie arbeiten, die langsamer als das Programm ist oder ...

Allerdings muss gesagt werden, dass es noch elegantere Methoden zur Verzögerungs-Programmierung gibt als die Lahmlegung des Computers, aber dazu kommen wir erst in einer späteren Folge.

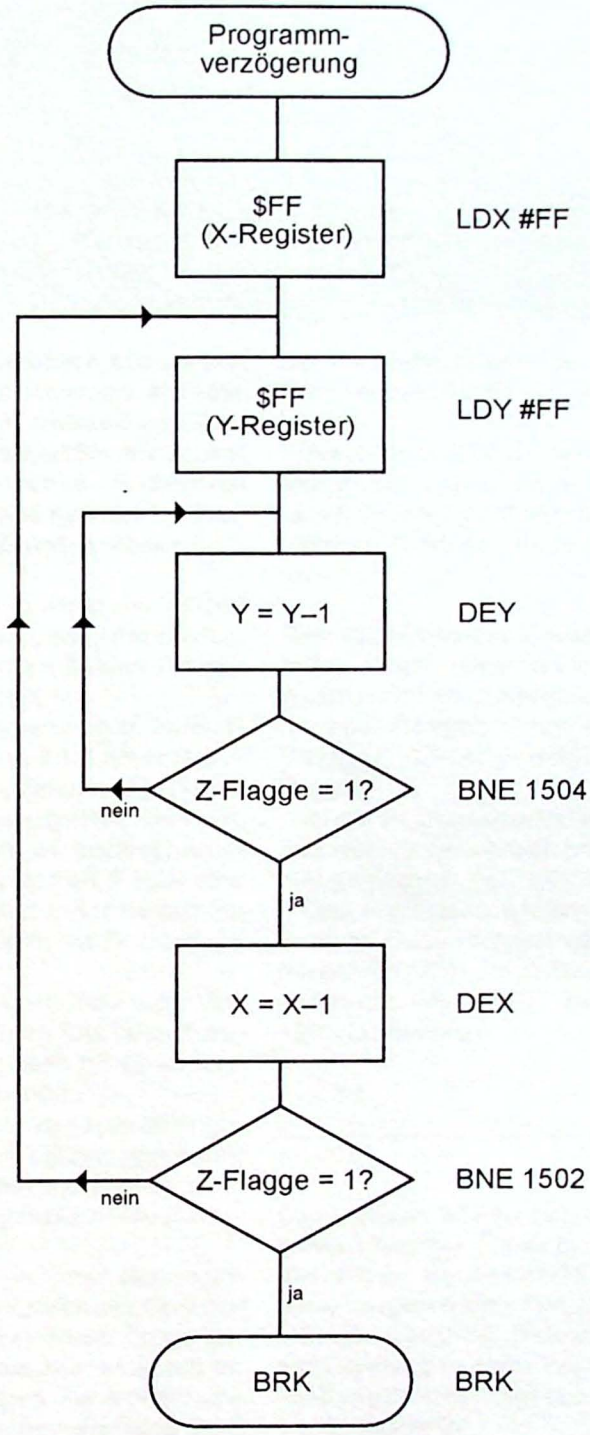


Abb. 7:
Flussdiagramm
zur Verzöge-
rungsschleife

Assembler ist keine Alchemie
Folge 4

Erschienen in: 64'er 12/1984
Verantwortlicher Redakteur: Albert Absmeier
Autor: Heimo Ponnath

In dieser Folge des Assembler-Kurses lernen Sie die wichtigen Arithmetik-Befehle des Prozessors kennen. Anhand von Beispielen und Übungen können Sie alle Schritte am Computer miterleben. Außerdem wird die Frage geklärt, wie Assembler-Programme in BASIC eingebunden werden.

Neun neue Befehle haben wir in der letzten Folge kennen gelernt und wir wissen nun, wie unser Computer ganze Zahlen (so genannte Integers) abspeichert.

Zur Erinnerung: Das geschieht im Zweierkomplement-Format. Das Bit 7 einer 8-Bit-Zahl dient dabei als Vorzeichen-Merkmal: Wenn es 0 ist, liegt eine positive Zahl vor, die genauso aussieht, wie wir bislang immer Binärzahlen kannten. Ist das Bit 7 aber eine 1, dann haben wir es mit einer negativen Zahl in der Zweierkomplement-Darstellung zu tun.

Wenn wir - wie unser Computer - zur Verarbeitung ganzer Zahlen 16 Bits (also 2 Bytes) verwenden, dann ist eben Bit 15 an Stelle von Bit 7 das Vorzeichenbit.

Wenn Sie nun am Ende der letzten Folge ein bisschen mit solchen Zahlen gerechnet haben, konnten Sie sicher feststellen, dass zwar oft das richtige Ergebnis herauskam - aber leider nicht immer.

Warum das so ist und was man deswegen noch beim Arbeiten mit Zahlen per Computer beachten muss, soll in dieser Folge behandelt werden. Damit wir aber nicht nur im vergleichsweise trockenen Zahlendschungel herumirren, sollen Sie heute endlich auch die wichtigsten Befehle des 6502 (beziehungsweise 6510) zur Arithmetik kennen lernen.

Außerdem gibt es dazu noch zwei Flaggen gratis, und die Branch-Befehle (schon lange überfällig) sollen Ihnen nun vertraut werden. Zunächst aber noch etwas Zahlensalat:

Herr Carry und der V-Mann

Keine Angst, wir sind nicht ins Krimi- oder Agentenmilieu gewechselt! Wir haben es mit zwei Flaggen zu tun, der Carry- und der V-Flagge. »To carry« heißt auf deutsch etwa »tragen«.

In der Registeranzeige ist diese Flagge immer mit »C« gekennzeichnet. Was wird denn hier getragen?

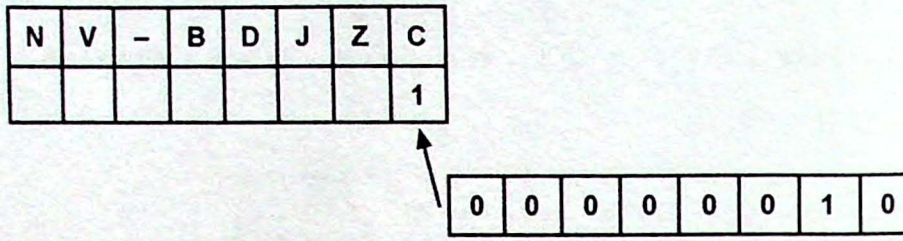
Das ergründen wir am besten an einem Beispiel. Dazu rechnen wir mit normalen Binärzahlen (also ohne Rücksicht auf Vorzeichenbits). Wir zählen die Zahlen 128 und 130 zusammen:

```
  128          1000 0000
+ 130        + 1000 0010
= 258        (1)0000 0010
```

Das Ergebnis 258 ist richtig - auch in der Binärdarstellung - nur es passt nicht mehr in 8 Bits. Ein Bit wurde überTRAGEN in ein extra dafür vorgesehenes Plätzchen:

in das Carry-Bit. Jedesmal also, wenn so ein Übertrag in einer Rechenoperation des C 64 stattfindet, zeigt die Carry-Flagge eine 1 (Abbildung 8).

Abb. 8: Das Carry-Bit als Bit 8 einer Rechenoperation



Je nach Art der von uns programmierten Aufgabe können wir nun dieses Carry-Bit weiter verarbeiten. Es gibt Situationen, in denen man es einfach ignorieren darf (dazu kommen wir gleich noch) oder aber solche, wo man es weiter in der Rechnung verwendet.

Schließlich kann es auch noch einen Fehler anzeigen: dann nämlich, wenn das größte zulässige Ergebnis 1111 1111 sein darf.

Natürlich kann das Carry-Bit auch gesetzt werden, wenn man in der Zweierkomplementform rechnet. Die Verhältnisse sind dann aber für ein leicht überschaubares Beispiel des Übertrags zu verwickelt, wie Sie gleich sehen werden.

Wenn wir nämlich mit den Zweierkomplement-Zahlen rechnen, dann interessieren uns auch Fälle wie bei der Addition von 64 und 66:

```
   64              0100 0000
 + 66            + 0100 0010
------          -----------
(- 126)          1000 0010
```

Das ist offensichtlich falsch. Bei der Addition ist durch das Zusammenzählen der Bits 6 plötzlich Bit 7 gesetzt worden. Da wir es aber mit einer Zweierkomplementzahl zu tun haben, bei der dieses Bit 7 eine negative Zahl anzeigt, folgt ein Fehler.

Es ist also von Bedeutung, so einen Überlauf (englisch: »overflow«) erkennen zu können, um eine entsprechende programmtechnische Reaktion zu starten. Es wird die Überlauf-Flagge V auf 1 gesetzt. Leider ist die Sache aber nicht so einfach, dass sie immer gesetzt würde, wenn von Bit 6 nach Bit 7 ein Übertrag stattfindet. Gesetzt wird diese V-Flagge nur in folgenden zwei Fällen:

1. Es findet ein Übertrag von Bit 6 nach Bit 7 statt, aber kein äußerer Übertrag (wie beim Carry).
2. Es findet kein interner Übertrag von Bit 6 nach Bit 7 statt, aber ein äußerer Übertrag passiert.

Merken kann man sich das am Besten so: Immer dann, wenn gewissermaßen das Vorzeichenbit 7 »versehentlich« verändert wurde, wird die V-Flagge auf 1 gesetzt.

Das ist ein harter Brocken! Wir sind es ja gewohnt, dass wir uns um diese Dinge beim Computer eigentlich gar nicht mehr kümmern müssen.

Außerdem würde das ja erfordern, dass man sich bei allen Operationen vorher überlegen muss, welche Zahlen auftreten können und welche Fehler daher durch »versehentliches« Vorzeichenändern passieren können!

Genauso ist es - in der Programmierpraxis wird Ihnen aber das ganze Problem nicht mehr so groß vorkommen.

Wir wollen uns dieses Zusammenspiel der Überträge von Bit 6 nach Bit 7 und von Bit 7 nach Bit 8 (also ins Carry-Bit) noch anhand einiger Beispiele klarer machen.

Im obigen Beispiel der Addition von 64 und 66 haben wir einen Fall schon behandelt: Es fand ein Übertrag von Bit 6 nach Bit 7 statt, aber kein äußerer Übertrag ins Carry-Bit. Deswegen wurde dann auch die V-Flagge gesetzt. Das Problem lässt sich hier ganz einfach lösen, zum Beispiel durch Verwendung von 16-Bit-Zahlen:

```
   64              0000 0000 0100 0000
 + 66            + 0000 0000 0100 0010
------          -----------------------
  130            0000 0000 1000 0010
```

Bei 16-Bit-Zahlen ist ja Bit 15 das Vorzeichenbit, welches hier keine Änderung erfährt.

Der andere Fall tritt auf bei der Addition von zwei negativen Zahlen wie -125 und -64:

```
 - 125                1000 0011
 -  64                1100 0000
(+  67)              (1)0100 0011
```

Auch das ist offensichtlich falsch: Es hat wieder »versehentlich« ein Vorzeichenwechsel stattgefunden. Dies ist also der Fall, wo zwar ein Übertrag ins Carry-Bit stattfand, aber kein Übertrag von Bit 6 nach Bit 7.
Auch dieses Problem lässt sich durch Verwendung von 16-Bit-Zahlen lösen. Eine kleine Trainingsaufgabe für Sie!

Man kann also sagen: Immer dann, wenn bei 8-Bit-Rechnungen der mittels Zweierkomplementzahlen darstellbare Bereich (127 bis -128) über- oder unterschritten wird, fuhrwerkt man im Vorzeichen-Bit herum und verfälscht das Ergebnis.

Dann leuchtet wie eine rote Ampel die Überlauf-(V-)Flagge auf und sagt uns, dass wir in diesen Fällen besser mit 16-Bit-Zahlen arbeiten sollten.

Nun noch zum Ignorieren des Carry-Bits, das ich weiter oben erwähnt habe. Bei allen 8-Bit-Rechenoperationen mit Zweierkomplementzahlen kann das Carry-Bit vernachlässigt werden.

Zwei Beispiele sollen das wieder illustrieren. Wir addieren +4 und -2:

```
+ 4                  0000 0100
- 2                  1111 1110
+ 2                (1)0000 0010
```

Das Carry-Bit wird außer acht gelassen. Anderes Beispiel: Wir addieren zwei negative Zahlen, -4 und -6:

```
−4                   1111 1010
−2                   1111 1110
−6                 (1)1111 1000
```

Auch hier kann man (sogar: muss man) das Carry-Bit vernachlässigen. Beide Ergebnisse sind richtig. Nun wissen Sie alles über die Art, wie unser Rechner mit ganzen Zahlen arbeitet. Probieren Sie mal zur Übung ein paar Aufgaben aus. Wir verlassen jetzt den Zahlendschungel und widmen uns der Praxis.

Der Computer rechnet: ADC, CLC

ADC ist der erste Arithmetik-Befehl des 6502 (und natürlich auch des 6510), den wir kennen lernen. Er bedeutet »ADd with Carry«, also »addiere mit Carry-Bit«. An einem 8-Bit-Beispiel wollen wir uns das mal ansehen. ZAHL1 und ZAHL2 sollen addiert werden. Beide sollen positive 8-Bit-Zahlen sein, die so klein sind, dass kein Überlauf zu erwarten ist. ZAHL1 wird in den Akku gegeben: `LDA #ZAHL1`

Wenn wir nun den Befehl `ADC #ZAHL2` folgen lassen, sorgt die ALU (arithmetisch-logische Einheit, siehe Folge 1) dafür, dass beide Zahlen addiert werden und das Ergebnis im Akku erscheint. ZAHL1 ist dann vom Ergebnis überschrieben worden.

An sich ist damit alles erledigt. Weil wir aber häufig wissen wollen, was denn nun bei der Addition herausgekommen ist, speichern wir den Akku-Inhalt noch irgendwo ab mittels »`STA Speicherstelle`«.

Außerdem war da ja noch die Sache mit dem Carry-Bit. Wir haben oben festgestellt, dass bei einer 8-Bit-Addition kein Carry-Bit berücksichtigt werden soll. Nun gibt es aber eine ganze Menge von Vorgängen in einem Assembler-Programm, die das Carry-Bit beeinflussen. Man kann eigentlich vor einer Addition nie ganz sicher sein, ob es denn nun 1 oder 0 ist. Weil jedoch ADC auch das Carry-Bit mit addiert, sollte man dafür sorgen, dass es vor dem Zusammenzählen wirklich gelöscht ist. Dazu gibt es den Befehl `CLC`, was die Abkürzung für »CLear Carry«, also »lösche Carry-Bit« ist. Wenn ZAHL1 = 12 und ZAHL2 = 7 ist, dann würde unser vollständiges 8-Bit-Additions-Prögrämmchen also lauten:

```
1200 CLC
1201 LDA #$0C
1203 ADC #$07
1205 STA 1500
```

Sehen wir mal davon ab, dass dieses Programm natürlich unsinnig ist (das kann man ja im Kopf schneller rechnen!), dann erkennen wir: CLC ist ein 1-Byte-Befehl mit impliziter Adressierung, welcher sich nur auf die C-Flagge (also das Carry-Bit) auswirkt. ADC ist in der hier verwendeten Form ein 2-Byte-Befehl und liegt in der »unmittelbar« genannten Adressierung vor. Wie wir oben gesehen haben, kann ADC - je nach Art der Rechnung - auf einige Flaggen wirken:

Da wären zunächst natürlich die V-Flagge und die C-Flagge. Dann aber kann beim Auftreten eines gesetzten Bit 7 auch die N-Flagge und beim Überschreiten von $FF eventuell auch die Z-Flagge verändert werden.

Viel interessanter wird unser Mini-Programm schon, wenn man an Stelle von

```
1201 LDA #$0C
```

jetzt die absolute Adressierung verwendet, zum Beispiel

```
1201 LDA 1400
```

Weil das ein 3-Byte-Befehl ist, verschiebt sich natürlich der Rest des Programms um 1 Byte. So kann man immerhin schon zu unterschiedlichen Inhalten von 1400 den gleichen Betrag addieren.

Am interessantesten allerdings ist die Tatsache, dass auch ADC absolut adressierbar ist. Wir können so zum Beispiel den Inhalt der Speicherzelle 1300 zum Inhalt der Zeile 1400 hinzu zählen und dann das Ergebnis in 1500 ablegen:

```
1200 CLC
1201 LDA 1400
1204 ADC 1300
1207 STA 1500
```

Hier ist der ADC-Befehl dann 3 Bytes lang geworden.

Vergessen Sie bitte nicht - das gilt vor allem für die nachfolgenden Rechenoperationen -, dann, wenn die Wahrscheinlichkeit besteht, dass der Dezimal-Modus eingeschaltet ist (also die D-Flagge auf 1 gesetzt ist), noch den Befehl CLD vor solche Programme zu stellen.

Solche 8-Bit-Rechnungen kommen recht häufig vor: Wenn man in Schleifen nicht mit mehrfach wiederholten INX (beziehungsweise INY oder INC, DEX, DEY oder DEC) arbeiten will, addiert man eben immer die Sprungweite mittels ADC hinzu. Der Akku kann nicht als Zähler dienen, denn es gibt für ihn keinen Befehl, der dem INX und so weiter vergleichbar wäre, weswegen man ihn - sollte es nötig sein - mittels ADC hoch zählt.

Häufiger und in der Praxis bedeutender sind 16-Bit-Rechnungen. Wie Sie sicher noch aus den vorangegangenen Folgen wissen, teilt man so eine 16-Bit-Zahl auf in zwei Bytes (das LSB und das MSB).

Nehmen wir für unser nachfolgendes Beispiel wieder an, dass die Zahlen so gebaut sind, dass kein Überlauf zu befürchten ist. ZAHL1 hätten wir vorher in die Speicherstellen 1300 (LSB) und 1301 (MSB) gepackt, ZAHL2 liegt in den Zellen 1400 (LSB) und 1401 (MSB). Zunächst wieder die Vorbereitungsmaßnahmen:

```
1200 CLD
1201 CLC
```

Dabei ist CLD nicht immer nötig, wie schon gesagt. Nun addieren wir zuerst die LSBs:

```
1202 LDA 1300
1205 ADC 1400
1208 STA 1500
```

Ein Überlauf kann hier nicht stattgefunden haben, denn das Vorzeichenbit ist ja im MSB als Bit 15 enthalten. Wohl aber kann ein Über-

trag stattgefunden haben: Das Ergebnis könnte größer als 255 ($FF) gewesen sein.

War das der Fall, dann ist jetzt eine 1 im Carry-Bit. Wir addieren nun die MSBs:

```
120B LDA 1301
120E ADC 1401
1211 STA 1501
```

Egal, was im Carry-Bit stand: Es wurde jetzt hinzu addiert. Das Ergebnis unserer Rechnung steht nun in 1500 (LSB) und 1501 (MSB). Sehen wir uns das Ganze noch mal am Zahlenbeispiel an. Wir addieren die Zahlen 2176 (binär: 0000 1000 1000 0000) und 1009 (binär: 0000 0011 1111 0001). Die Speicherinhalte sind dann:

```
1300    1000 0000    LSB Zahl1
1301    0000 1000    MSB
1400    1111 0001    LSB Zahl2
1401    0000 0011    MSB
```

Jetzt addieren wir die LSBs:

```
1300    1000 0000
1400    1111 0001
Carry:         0
-------------------
1500    0111 0001
Carry:         1
```

Nun folgt der zweite Teil der Addition mit den MSBs:

```
1301    0000 1000
1401    0000 0011
Carry:         1
-------------------
1501    0000 1100
```

Damit steht nun das Ergebnis 3185 (binär 0000 1100 0111 0001) säuberlich aufgeteilt in LSB (Speicher 1500) und MSB (Speicher 1501) fest. Das Carry-Bit steht auch nach vollendeter Rechnung noch auf 1, so dass es vor erneuter Addition wieder mit CLC zu löschen ist. Damit wäre alles über die Addition berichtet. Wie immer in Programmiererkreisen die Empfehlung: üben, üben ... Wir wenden uns jetzt der gegenläufigen Operation zu: der Subtraktion.

Noch mehr Rechnen: SBC, SEC

Dass das Abziehen von Zahlen im Computer durch das Hinzuzählen des Zweierkomplements geschieht, haben wir mit viel Gehirnschmalzverbrauch schon in vorangegangenen Abschnitten erfahren. Nun sollen Sie die dazu nötigen Befehlsworte des Assemblers kennen lernen. Zunächst einmal ist da SBC. Das heißt »SuBtract with Carry« oder auf deutsch etwa »ziehe unter Berücksichtigung des Carry-Bits ab«.

Ebenso wie bei der Addition mit ADC wirkt das Argument des SBC-Befehls auf den Akku-Inhalt ein - wobei das Ergebnis im Akku landet, diesen also überschreibt.

Komplizierter ist hier die Verwendung des Carry-Bits, worauf wir aber nicht detailliert eingehen wollen. (Wen es interessiert: nachlesen in L. A. Leventhal, »6502 programmieren in Assembler«, 3. Auflage, München 1983, Seiten 3 - 100).

Für uns soll einfach die nicht ganz korrekte Analogie zum »Borgen« bei der Subtraktion ausreichen. Für den Fall, dass ein solches Borgen eintreten muss, sollte auch das dazu nötige Carry-Bit vorhanden sein (also auf 1 gesetzt sein). Wie Sie sicherlich schon erraten haben, heißt SEC »SEt Carry«, also »setze das Carry-Bit« (auf 1).

Merke: Vor einer Addition immer Löschen des Carry-Bits mit CLC, vor einer Subtraktion immer Setzen des Carry-Bits mit SEC!

Zwei Beispiele für die Subtraktion sollen das bisher Gesagte erläutern. Zunächst eine 8-Bit-Subtraktion von ZAHL1 (in Speicherzelle 1300) minus ZAHL2 (in Zelle 1400). Ds Ergebnis wird nach 1500 geschrieben:

```
1200 CLD
1201 SEC
1202 LDA 1300
1205 SBC 1400
1208 STA 1500
```

SBC kann - wie hier - absolut adressiert werden, aber auch unmittelbar (also zum Beispiel SBC $40).

Der Befehl ist dann im ersten Fall ein 3-, im anderen Fall ein 2-Byte-Befehl. SEC ist ebenso wie vorher schon CLC ein implizit adressierbarer 1-Byte-Befehl.

Das zweite Beispiel ist eine 16-Bit Subtraktion. In den Speichern steht vor dem Aufruf dieser kleinen Routine:

```
1300 ZAHL1    LSB
1301 ZAHL1    MSB
1400 ZAHL2    LSB
1401 ZAHL2    MSB
```

Das Ergebnis soll nach 1500 (LSB) und 1501 (MSB) gebracht werden:

```
1200 CLD
1201 SEC
1202 LDA 1300
1205 SBC 1400
1208 STA 1500
```

Jetzt sind die beiden LSBs von einander abgezogen und die Differenz abgespeichert als LSB des Ergebnisses.

```
120B LDA 1301
120E SBC 1401
1211 STA 1501
```

Damit ist die Aufgabe beendet. Auch die MSBs sind subtrahiert und das MSB des Ergebnisses steht in 1501. SBC beeinflusst die gleichen Flaggen wie der Befehl ADC.

Ein Programmprojekt

Damit die so kennen gelernten Arithmetik-Befehle nicht so trocken auf weiter Flur stehen, wollen wir nun ein Programm entwickeln, aus dem zweierlei zu lernen ist:

1. die Anwendung bisher gelernter Befehle
2. ein häufig angewendetes Verfahren, Assemblerprogramme in BASIC-Programme einzubinden

Besonders dieser zweite Aspekt scheint noch vielen Lesern unklar zu sein (das zeigen mir Zuschriften). Es gibt eine ganze Reihe von Möglichkeiten zum Einbau von Assembler-Routinen in BASIC-Programme; die werden wir alle nach und nach kennen lernen.

Von Ihnen sicherlich schon häufig angewendet wurde der SYS-Befehl (zum Beispiel für SYS 58640 und vorherigem POKE 214, Zeile und POKE 211,Spalte zum Setzen des Cursors an die Stelle Zeile,Spalte).

Damit haben Sie ein Maschinenprogramm aufgerufen, das im System unseres Computers schon enthalten ist. 58640 ist die Startadresse des Programms, und man kann diesen SYS-Befehl eigentlich wie eine Art »GOTO Maschinenprogramm-Startadresse« ansehen.

Nichts hindert uns also, auf diese Weise eigene Assembler-Programme anzuspringen! Das Problem liegt nun nur noch darin, wie man Parameter, die unsere Maschinenroutine benötigt, übergeben kann. Eine offensichtliche - aber leider auch relativ langsame - Methode ist das POKEn der Werte im LSB / MSB-Format in die Speicherzellen, aus denen sie sich unser ML-Programm dann abholt. Wir wollen dieses Verfahren nun an einem Programmbeispiel verwenden.

Eine arithmetische Reihe werden viele von Ihnen schon kennen. Wenn man A als erstes Glied, D als Differenz und N als die Anzahl der Glieder bezeichnet, dann ist die Summe einer solchen Reihe:

S = A + (A + D) + (A + 2 * D) + ... usw ... + (A + (N - 1) * D)

Ein Beispiel ist die Summe der ersten zehn ganzen Zahlen:

S = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10

Hier ist A = 1, D = 1 und N = 10. Dass die Summe S im Beispiel 55 ist, kann man schnell berechnen - was aber, wenn wir wesentlich mehr als nur zehn Glieder haben?

Es gibt natürlich auch Formeln zur Berechnung von S. Aber eigentlich ist es ganz reizvoll, ohne solche Formeln den Computer die Summe bilden zu lassen. Wir bauen also ein Programm zur Berechnung der Summe der ersten N ganze Zahlen, wobei N frei gewählt werden kann. Das Ergebnis soll eine 16-Bit-Zahl sein, also nicht größer als 32 767. Das beschränkt uns bei N auf Werte von 1 bis 255 (warum, können Sie ja mal mit dem fertigen Programm ausprobieren).

N benötigt also nur 1 Byte Speicherplatz und soll in $1300 abrufbar sein. A soll 1 sein, ebenso wie D. Für eventuelle Programmänderungen ist es aber sinnvoll, A und D als 16-Bit-Zahlen aufzubewahren – und zwar in $1310/ 1311 (A im LSB / MSB-Format) und in $1320/ 1321 (D im gleichen Format).

Das Ergebnis soll in $1400/ 1401 zu finden sein. Das Maschinenprogramm legen wir nach $1200. Zuerst kümmern wir uns um das BASIC-Aufrufprogramm:

```
10 REM AUFRUF SUMME ARITHMETISCHE REIHE
20 POKE5120,0 : POKE5121,0 : REM ERGEB-
NISSPEICHER AUF NULL
30 PRINT CHR$(147) CHR$(17) CHR$(17)
40 INPUT »ANZAHL DER GLIEDER N = ";N
50 IF N < 1 OR N > 255 THEN PRINT
CHR$(17)"1 < = N <=255" : GOTO 40
60 POKE 4864,N : REM EINSPEICHERN VON N
70 POKE 4880,1 : POKE4881,0 : POKE
4896,1 : POKE4897,0 : REM EINSPEICHERN
VON A UND D
80 SYS 4608 : REM AUFRUF UNSERES MASCHI-
NENPROGRAMMS
90 M = PEEK (5121) : L = PEEK(5120) :
REM AUSLESEN DES ERGEBNISSES
100 E = 256*M + L : PRINTCHR$(17)CHR$(17)
110 PRINT "DIE SUMME DER ERSTEN N GANZEN
ZAHLEN IST:" : PRINT E
120 END
```

Zu diesem Programm gibt es nur noch zu bemerken, dass die Zahlen bei POKE, PEEK oder SYS die Dezimalwerte unserer oben gewählten Adressen sind.

Nun endlich zum Assemblerprogramm. Sehen Sie sich dazu bitte das Flussdiagramm in Abbildung 9 an.
Wir bereiten den Ablauf vor, indem wir aus $1300 die Anzahl der Glieder ins X-Register

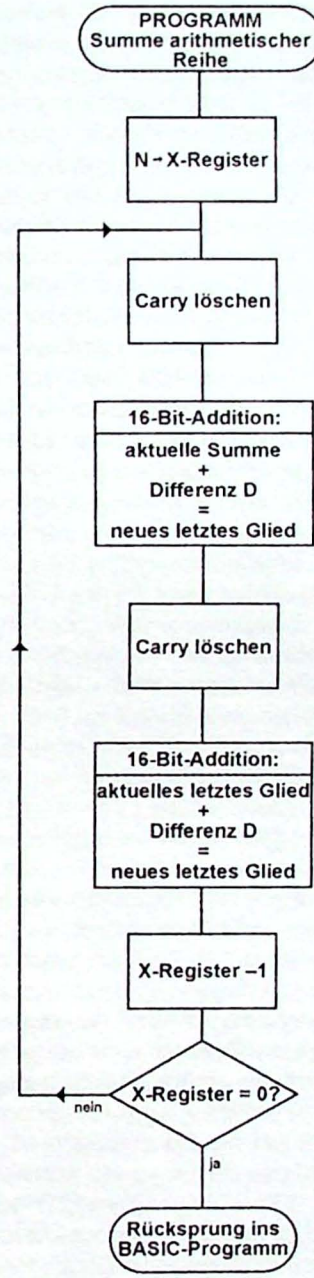


Abb. 9: Flussdiagramm des Assemblerprogramms »Summe einer arithmetischen Reihe«

laden und zur Vorbereitung der Addition das Carry-Bit löschen. Schalten Sie also bitte den SMON ein und tippen Sie A 1200 <RETURN>. Es erscheint die Startadresse 1200. Jetzt können Sie Zeile für Zeile das Assembler-Programm eingeben (nach jeder Zeile ein <RETURN>, das die nächste Zeilennummer erzeugt):

```
1200 LDX 1300
1203 CLC
```

Die nächsten sechs Zeilen summieren jeweils das neueste Glied zur bis dahin erzeugten Summe. Zu Beginn ist $1400 / 1401 noch leer und in $1310 / 1311 steht noch das Anfangsglied A = 1. Später, mit Durchlaufen der Schleife, steht in $1400 / 1401 immer die bis dahin gebildete Summe und in $1310 / 1311 das letzte Glied der Reihe. Es handelt sich um die oben kennen gelernte 16-Bit-Addition:

```
1204 LDA 1400
1207 ADC 1310
120A STA 1400
```

Das neue LSB ist berechnet und in $1400 geschrieben.

```
1200 LDA 1401
1210 ADC 1311
1213 STA 1401
```

Das war nun noch das neue MSB. Als Nächstes berechnen wir das momentan letzte Glied der Reihe durch Addieren von D zum alten letzten Glied.

Das entspricht dem BASIC-Befehl A = A + D in einer Schleife. Dies ist eine neue 16-Bit-Addition, weshalb wir wieder CLC vorgeben müssen:

```
1216 CLC
1217 LDA 1310
121A ADC 1320
121D STA 1310
```

Das war wieder das LSB. Nun zum MSB:

```
1220 LDA 1311
1223 ADC 1321
1226 STA 1311
```

Wir zählen nun das X-Register um 1 herunter und prüfen, ob es schon Null geworden ist, ob also schon alle N-Glieder summiert worden sind:

```
1229 DEX
122A BNE 1203
```

Wenn noch nicht alle Glieder berechnet und summiert sind, kehren wir an den Schleifenanfang zurück. Ansonsten springen wir zurück ins BASIC-Aufrufprogramm: 122C RTS

Wenn Sie beide Programme eingetippt haben, dann speichern Sie sie vorsichtshalber ab (das Assemblerprogramm mit dem S-Befehl des SMON). Beim neuen Einladen brauchen Sie den SMON nicht mehr.

Nach dem Laden unseres Maschinenprogramms (mit »,8,1« bei Diskette oder »,1,1« bei Kassette) geben Sie »NEW« < RETURN > ein, damit die Zeiger vor dem Einladen des BASIC-Programms wieder auf Normalwerte gesetzt werden. Zwischen dem dann eingeladenen BASIC-Programm und unserer Assembler-Routine ist genug Platz. Sollten Sie aber irgendwann mal das BASIC-Programm vergrößern, schützen Sie bitte unseren Bereich ab $1200.

Unser Assembler-Beispiel ist so aufgebaut, dass auch A und D variabel gehalten sind. Sie müssten dann nur noch Eingabemöglichkeiten im BASIC-Programm schaffen und an Stelle der Werte 1 oder 0 in Zeile 70 die LSBs und MSBs der von Ihnen eingegebenen Größen A und D einPOKEn.

Auf diese Weise sind dann beliebige ganzzahlige, arithmetische Reihen berechenbar, wie zum Beispiel S = 7 + 10 + 13 + 16 + ... und so weiter. Das überlasse ich Ihrer BASIC-Programmierfertigkeit.

Nur Eines noch: Sie müssen darauf achten, dass die Summe S nicht größer als 32767 wird. Ihrer Fantasie sind - wie immer in diesem Metier - keine Grenzen gesetzt. Sie könnten sich ja mal überlegen, wie man größere Summen zulassen kann (wer sagt denn, dass wir Zahlen immer nur in 2 Bytes darstellen dürfen?).

Oder Sie könnten sich überlegen, welches eindeutige Merkmal auftritt, sobald der Maximalwert überschritten wird (ein Tipp: Lesen Sie doch mal den Abschnitt über die V-Flagge nach).

Die Branch-Befehle

Der 6502 (und auch der damit identische 6510) kennt acht bedingte Verzweigungen, von denen wir bisher BNE schon verwendet haben.

Alle diese Branch-Befehle (von »branch« = verzweigen) prüfen Flaggen des Statusregisters.

BNE und BEQ beziehen sich auf die Z-Flagge, die anzeigt,ob im Verlauf der letzten Operation eine Null aufgetreten ist. Ist das der Fall, steht in der Z-Flagge eine 1.

BNE verzweigt zur angegebenen Adresse, wenn in der Z-Flagge eine 0 enthalten ist. BEQ (»Branch if EQual zero« = verzweige, wenn gleich Null) tut das dann, wenn die Z-Flagge auf 1 gesetzt ist. Da muss man etwas aufpassen, damit man sich nicht vertut!

BCC und BCS haben ihre Aufmerksamkeit auf die C-Flagge, also das Carry-Bit gerichtet. BCC kommt vom englischen »Branch if Carry Clear«, was heißt: »verzweige, wenn das Carry-Bit gelöscht ist«. Ein gesetztes Carry-Bit (also Inhalt = 1) veranlasst BCS (»Branch if Carry Set« = verzweige, wenn das Carry-Bit gesetzt ist) zum Sprung an die angegebene Adresse.

Diese vier bedingten Verzweigungen sind an sich die bedeutsamsten und am häufigsten verwendeten Branch-Befehle. Man kann wohl getrost sagen, dass über 90 Prozent der von Programmierern verwendeten bedingten Sprünge damit absolviert werden.

R. Mansfield warnt in seinem Buch »Machine Language for Beginners«, einem in den USA sehr verbreiteten Werk, sogar ausdrücklich vor der Verwendung der Befehle BPL und BMI!

Dafür liegt absolut kein einsehbarer Grund vor. Viele programmtechnische Aufgabenstellungen lassen sich elegant und leicht mit BPL, BMI, BVS und BVC lösen. Man muss nur wissen, wie sie funktionieren und - da liegt vermutlich der Hund begraben - man muss auch die Art kennen, wie Zahlen vom Computer behandelt werden.

Genau das aber wissen wir, und deswegen sollten wir diese Kenntnis für uns auch nutzen. Also ohne Scheu heran an die verfemten Befehle!

BMI und BPL (Branch on MInus = verzweige, wenn negativ und Branch on PLus = verzweige, wenn positiv) hängen mit der Negativ-Flagge N zusammen. Das Rätsel dieser Flagge konnte in den vorangegangenen Folgen gelöst werden: Immer dann, wenn bei einer Operation eine Zahl auftrat, deren Bit 7 eine 1 war, wurde die N-Flagge auf 1 gesetzt. Wir wissen jetzt, dass dieses Bit bei 8-Bit-Zahlen das Vorzeichenbit ist.

Bit 7 sagte uns bei einer 1, dass eine negative Zahl im Zweierkomplement-Format vorliegt oder aber überhaupt ein Speicherzelleninhalt vorhanden ist, der größer als 0111 1111 = 127 ist. BMI führt zum Sprung in diesem Fall, weil die N-Flagge auf 1 steht. Andernfalls führt BPL zur Verzweigung.

Ebenso einfach sind BVS und BVC zu erklären: Sie beziehen sich auf die V-Flagge, unsere rote Ampel, die Überlauf bei Rechenoperationen anzeigt.

Kann es was Bequemeres geben zur Behandlung solcher Fehlrechnungen als ein »Branch on oVerflow Set« = »verzweige, falls die Überlauf-Flagge gesetzt (= 1) ist« mit BVS? Oder anders herum bei BVC »Branch on oVerflow Clear« = »verzweige bei freier Überlauf-Flagge«?

Wenn man - wie Sie jetzt nach dieser Folge - weiß, unter welchen Umständen diese

V-Flagge auf 1 gesetzt wird, sollte man BVS und BVC ohne Skrupel ausgiebig benutzen. Man könnte damit zum Beispiel programmieren, dass die Rechengenauigkeit automatisch von 16 Bit auf 24 oder 32 (oder wie es gerade beliebt) Bit gesteigert wird, ohne dass man sich bei jeder Programmaufgabe Gedanken über das größtmögliche Ergebnis machen muss. Dazu aber ein andermal mehr.

Alle hier vorgestellten Branch-Befehle sind ebenso wie BNE 2-Byte-Befehle, was an der speziellen Art der Adressierung liegt: der relativen Adressierung.

Eigentlich hatte ich Ihnen ja versprochen, diese relative Adressierung zusammen mit den Branch-Befehlen zu erklären. Ich werde ihr aber lieber einen eigenen Abschnitt widmen, weil's zum genauen Verständnis doch etwas mehr Aufwand braucht. Die nächste Folge fängt dann damit an, abgemacht?

Wie die anderen Folgen auch, soll auch diese hier noch mit einer Tabelle enden, in der die neu gelernten Befehle mit Zubehör gezeigt sind.

Tab. 7: die 11 neuen Befehle

Befehlswort	Adressierung	Byte-Anzahl	Kode		Dauer in Taktzyklen	Beeinflussung von Flaggen
			HEX	DEZ		
ADC	unmittelbar	2	69	105	2	N, V, Z, C
	absolut	3	6D	109	4	N, V, Z, C
CLC	implizit	1	18	24	2	0 bis C
SBC	unmittelbar	2	E9	233	2	N, V, Z, C
	absolut	3	ED	237	4	N, V, Z, C
SEC	implizit	1	38	56	2	1 bis C
BEQ	relativ	2	F0		2	keine Änderung
BCC	relativ	2	90		2	keine Änderung
BCS	relativ	2	B0		2	keine Änderung
BMI	relativ	2	30		2	keine Änderung
BPL	relativ	2	10		2	keine Änderung
BVC	relativ	2	50		2	keine Änderung
BVS	relativ	2	70		2	keine Änderung
					bei relativer Adressierung: +1 bei Verzweigung +2 bei Überschreiten einer Seitengrenze	

Assembler ist keine Alchemie
Folge 5

Erschienen in: 64'er 1/1985
Verantwortlicher Redakteur: Albert Absmeier
Autor: Heimo Ponnath

In dieser Folge des Assembler-Kurses wird die relative Adressierung erklärt. Damit verbunden sind auch die wichtigen Vergleichsoperationen. Anhand einer sehr häufig verwendeten Betriebssystem-Routine können Sie Ihr neu erworbenes Wissen testen.

In der letzten Folge haben wir die Branch-Befehle kennen gelernt. Heute wollen wir uns mit der relativen Adressierung dieser Befehle und noch einer anderen Art der Adressierung befassen.

Weiterhin werden Sie einige neue Assembler-Worte lernen, nämlich die Vergleichsbefehle. Wie ganze Zahlen im Computer gespeichert sind, wissen wir bereits.

Heute untersuchen wir die Speicherung von Zeichen. Schließlich werden wir unsere Nase noch ein wenig in die eingebaute Software des C 64 stecken.

Die relative Adressierung

Als wir den BNE-Befehl das erste Mal verwendet haben, stellten wir fest, dass zum Beispiel `BNE 1200` nicht - wie eigentlich zu erwarten war - ein 3-Byte-Befehl, sondern ein 2-Byte-Befehl ist. Damals mussten wir uns mit der Bemerkung zufrieden geben, es läge an der besonderen Art der Adressierung, nämlich der relativen Adressierung.

Relativ bedeutet ja »bezogen auf etwas«. Wenn wir also beispielsweise `BNE 1200` schreiben, liegt es nur an der Benutzerfreundlichkeit des SMON und vieler anderer Assembler, dass dieser die so geschriebene absolute Adresse 1200 in die richtige Form, nämlich die relative, umrechnet. In Wahrheit verlangt der 6502 (und natürlich ebenso der 6510) eine Angabe darüber, wie viele Bytes nach vorne oder hinten im Programm er zur weiteren Programmverarbeitung springen (verzweigen) soll. Es gilt nun also, zwei Fragen zu klären:

1. Relativ wozu wird gesprungen?
2. Wie berechnet sich die Angabe, um wie viele Bytes nach vorne oder hinten im Programm der Sprung vollzogen werden soll?

Zur Klärung verwenden wir ein hypothetisches Programmsegment mit einem Sprungbefehl und sehen uns das Disassembler-Listing an:

```
2000 AD 00 30    LDA 3000
2003 F0 05       BEQ 200A
2005 A9 00       LDA #00
2007 8D 00 30    STA 3000
200A 60          RTS
```

Dieses Programm-Teilchen lädt den Inhalt der Speicherstelle 3000 in den Akku, überprüft dann, ob dieser Inhalt Null ist und verzweigt beim Vorliegen der Null zum Rücksprung (`RTS`). Ist der Inhalt von 3000 nicht Null, dann wird 3000 auf Null gesetzt. 3000 könnte zum Beispiel eine Flagge sein.

Der Pfad, dem der Computer bei der Abarbeitung des Programms folgt, wird durch den Programmzähler vorbereitet. Dieser ist

dann, wenn der BEQ-Befehl an der Reihe ist, schon einen Schritt weiter - im Programmzähler steht dann nämlich die Adresse 2005.

Relativ zu dieser Adresse hat dann der Sprung zu erfolgen. Zum Inhalt des Programmzählers muss also die Sprungweite (häufig auch Offset genannt) addiert werden. Soweit zur Frage 1.

Zur Klärung von Frage 2 listen wir uns mal Byte für Byte unser Programm auf:

```
Byte      Inhalt   Bedeutung
2000      AD       LDA
2001      00       LSB von 3000
2002      30       MSB von 3000
2003      F0       BEQ
2004      05       Offset
2005      A9       LDA #
2006   1  00
2007   2  8D       STA
2008   3  00       LSB von 3000
2009   4  30       MSB von 3000
200A   5  60       RTS
```

Neben der Byte-Nummer ist noch die Entfernung zu 2005 geschrieben. Daraus ist deutlich zu erkennen, dass die Sprungweite, die zum Programmzähler addiert wird, 05 sein muss, wenn der Sprung zum RTS erfolgen soll.

Für Vorwärts-Verzweigungen gilt also: Von der Adresse des Befehls an, der auf den Branch-Befehl folgt, zählt man die Byte-Anzahl bis zum Sprungziel. Das Ergebnis ist der Offset.

Nun gibt es genauso häufig Rückwärts-Sprünge. In den bisher gezeigten Programmen sind sie mehrmals aufgetreten. Wie berechnet man den Offset in diesen Fällen?

Sehen wir uns wieder das Disassembler-Listing eines solchen Programmsegments an:

```
1000    A2 00    LDX #00
1002    E8       INX
1003    D0FD     BNE 1002
1005    00       BRK
...
```

Dieses Prográmmchen tut nichts anderes, als das vorher auf Null gesetzte X-Register hoch zu zählen, bis es über 255 läuft (dann tritt ja wieder 0 auf!).

Solange der Inhalt des X-Registers ungleich Null ist, erfolgt ein Sprung zurück bis zur INX-Anweisung in Zeile 1002. Erst wenn die Null durch den Überlauf aufgetreten ist, endet das Programm mit einem BRK in Zeile 1005.

Wir wissen schon, dass der Programmzähler beim Verarbeiten des BNE-Befehls auf 1005 steht. Sehen wir uns auch dieses Programm Byte für Byte an:

```
Byte      Inhalt   Bedeutung
1000      A2       LDX #
1001      00
1002   3  E8       INX
1003   2  D0       BNE
1004   1  FD       Offset
1005      00       BRK
```

Wieder ist neben der Bytenummer die Entfernung vom aktuellen Programmzählerstand angegeben. Wir müssen also vom Inhalt des Programmzählers 3 abziehen, um zum INX-Befehl in Byte 1002 zu gelangen.

Das kennen wir aber schon aus den vergangenen Ausgaben: Wenn der Computer eine Zahl abzieht, dann addiert er das Zweierkomplement dieser Zahl. Hier soll nun 3 subtrahiert werden. Wir berechnen das Zweierkomplement:

```
 3  =  0000 0011  (binär)
       1111 1100  (Einerkomplement)
+1     1111 1101  (Zweierkomplement)
```

Dies ist das Zweierkomplement. Hexadezimal ausgedrückt heißt diese Zahl $FD und ist unser Offset. Für Rückwärts-Verzweigungen gilt also: Von der auf die Branch-Anweisung folgenden Speicherstelle an zählt man die Bytes zurück bis zum Sprungziel. Das Zweierkomplement der sich dadurch ergebenden Byte-Anzahl ist der Offset.

Das sieht reichlich kompliziert aus, aber zum Einen haben Sie ja einen ganz freundlichen Assembler, und nur in seltenen Notfällen müssen Sie den Offset berechnen. Zum Anderen gibt es noch eine Faustregel, mit der man das Ganze vereinfachen kann. Die soll durch folgendes Schema erläutert werden:

```
Byte   Inhalt                 Offset
...
1995                            F9
1996                            FA
1997                            FB
1998                            FC
1999                            FD
2000   BNE                      FE
2001   Offset                   FF
2002   Programmzählerstand
2003                            01
2004                            02
2005                            03
```

Bei Vorwärtssprüngen ist ohnehin alles klar: Bei einem Sprung nach Adresse 2005 müsste man in vorliegendem Fall einen Offset von 03 eingeben.

Bei Rückwärts-Verzweigungen zählt man einfach von $FF an rückwärts bis zur Zieladresse. Eine Verzweigung nach 1996 würde im vorliegenden Fall also einen Offset von $FA erfordern.

Eine Einschränkung der relativen Adressierung können Sie nun auch sofort verstehen, wenn Sie an Zweierkomplementzahlen denken: Der Offset belegt ein Byte. Die größte positive Zahl in einem Byte ist 0111 1111 = +127 = $7F und die kleinste negative Zahl ist 1000 0000 = -128 = ($80).

Es sind keine größeren Vorwärts-Verzweigungen als um 127 Bytes möglich, weil in diesem Fall ein Offset größer als $7F, also mit einem Bit 7 gleich 1 nötig wäre, was aber wieder als negative Zweierkomplementzahl verstanden und einen Rückwärtssprung verursachen würde.

Ähnliches gilt anders herum: Es ist kein weiterer Rücksprung als um 128 Bytes möglich, weil das im Offset zum gelöschten Bit 7 führen würde, also zu einem Offset kleiner als $80, was wiederum an Stelle des Rücksprunges eine Vorwärts-Verzweigung herbeiführen würde.

Darauf sollte man beim Erstellen eines Assembler-Programms achten: nie weitere Rückwärtssprünge als um 128 beziehungsweise Vorwärtssprünge um 127 Bytes verlangen!

Auch wenn man im Assembler gar nicht auf relative Adressierung Rücksicht nehmen muss, weil der Assembler sich mit den Absolutadressen begnügt, sollte man wissen, dass zum Beispiel folgende Zeile aufgrund dieser Einschränkung nicht möglich ist:

```
3000 BNE 1000
```

Die meisten Assembler reagieren auf solch eine Zeile mit einer Fehlermeldung oder so wie der SMON, der klammheimlich die Programmstartadresse statt 1000 einsetzt.

Aber es ist doch ärgerlich, wenn man auf dem Papier ein Programm fertig hat und erst beim Eintippen feststellt, dass der Computer das so nicht haben will.

Zeropage-Adressierung

Weil wir nun gerade mit der Adressierung so schön in Schwung sind, stelle ich Ihnen noch eine andere vor: die Adressierung der Zeropage.

Was ist die Zeropage? Auf deutsch heißt das Nullseite. Am besten versteht man das, wenn man sich in Erinnerung ruft, wie Adressen in unserem Computer verwaltet werden. Da haben wir doch ein LSB (Least Significant Byte) und ein MSB (Most Significant Byte), zum Beispiel $1F 04 (mit 1F als MSB und 04 als LSB).

Nun hat unser C 64 65 535 Adressen von $0000 bis $FFFF. Bei den ersten 256 Adressen von $0000 bis $00FF ist das MSB $00. Man nennt so einen 256-Byte-Block eine Seite (engl. page). Weil hier für alle Adressen dieser ersten Seite das MSB Null ist,

heißt sie Nullseite = Zeropage. Messerscharf werden Sie schließen, dass man die Seite mit den MSBs $01 als erste Seite bezeichnet, die mit den MSBs $02 als zweite Seite und so weiter.

Wenn wir nun zum Beispiel den Akku mit dem Inhalt der Zeropage-Adresse $00 FA laden wollen, dann könnten wir schreiben:

```
3000 LDA 00FA
```

Unser Mikroprozessor versteht uns aber auch, wenn wir nur schreiben: 3000 LDA FA. Das ist sie, die Zeropage-Adressierung. An Stelle eines 3-Byte-Befehls ist das jetzt ein 2-Byte-Befehl, was Speicherplatz und vor allem Rechenzeit einspart. Auf diese Weise kann man von den bisher kennen gelernten Befehlen Folgende adressieren: LDA, LDX, LDY, STA, STX, STY, INC, DEC, ADC und SBC. Sie können sich merken, dass man (bis auf zwei Ausnahmen, die wir noch kennen lernen werden) alle absolut adressierbaren Befehle auch Zeropage-absolut anwenden kann.

Genauere Angaben über die Kodes, die Ausführungszeiten und die Beeinflussung der Flaggen (Letztere ist identisch mit der absoluten Adressierung) entnehmen Sie bitte Tabelle 8.

Tab. 8: Kenndaten der neuen Befehle und Adressierungen

Befehlswort	Adressierung	Byte-Anzahl	Kode		Dauer in Taktzyklen	Beeinflussung von Flaggen
			HEX	DEZ		
LDA	0-Page, absolut	2	A5	165	3	N, Z
LDX	0-Page, absolut	2	A6	166	3	N, Z
LDY	0-Page, absolut	2	A4	164	3	N, Z
STA	0-Page, absolut	2	85	133	3	-
STX	0-Page, absolut	2	86	134	3	-
STY	0-Page, absolut	2	84	132	3	-
INC	0-Page, absolut	2	E6	230	5	N, Z
DEC	0-Page, absolut	2	C6	198	5	N, Z
ADC	0-Page, absolut	2	65	101	3	N, V, Z, C
SBC	0-Page, absolut	2	E5	229	3	N, V, Z, C
CMP	unmittelbar	2	C9	201	2	N, Z, C
	absolut	3	CD	205	4	N, Z, C
	0-Page, absolut	2	C5	197	3	N, Z, C
CPX	unmittelbar	2	E0	224	2	N, Z, C
	absolut	3	EC	236	4	N, Z, C
	0-Page, absolut	2	E4	228	3	N, Z, C
CPY	unmittelbar	2	C0	192	2	N, Z, C
	absolut	3	CC	204	4	N, Z, C
	0-Page, absolut	2	C4	196	3	N, Z, C

Zum Thema Geschwindigkeit: Wenn Sie die benötigten Taktzyklen von absolut und von 0-absolut adressierten Befehlen in den Tabellen miteinander vergleichen, werden Sie jeweils einen Unterschied von einem Zyklus feststellen.

Das mag Ihnen läppisch vorkommen. Bedenken Sie aber, dass Sie sehr häufig Schleifen programmieren müssen, die mehrere 100-mal durchlaufen werden, die vielleicht als oft zu verwendende Unterprogramme dienen ... Sie werden bald feststellen, dass da schnell beachtliche Zeitunterschiede auftreten können: Für zeitkritische Programme ist die Verwendung der Zeropage-Adressierung dringend geboten.

Dieser Tatsache waren sich leider auch die Schöpfer unseres Betriebssystems und des BASIC-Interpreters voll bewusst. Die Zeropage ist nahezu randvoll mit Speicherstellen, in denen sich beide Programmkomplexe tummeln.

Fast jede Kernal- und Interpreter-Routine notiert sich irgendwelche Werte auf der Seite Null. Das macht es uns als Assembler-Programmierer nicht gerade leicht, die Zeropage-Adressierung zu verwenden, wenn wir außerdem den Interpreter oder das Betriebssystem benutzen wollen.

Es kann geradezu katastrophale Folgen haben, einige Zeropage-Adressen zu überschreiben. Andere werden ständig durch das Betriebssystem oder den Interpreter neu beschrieben , was unseren eigenen - vielleicht gerade in so einer Speicherzelle gelagerten - Zwischenwerten den Garaus machen würde.

Man sollte sich also die ersten 256 Speicherstellen ganz genau ansehen, bevor man sie adressiert oder aber auf das Betriebssystem und den BASIC-Interpreter verzichten.

Ersteres erleichtern uns Tabellen der Speicherbelegung (zum Beispiel Babel, Krause, Dripke: »Das Interface-Age-Systemhandbuch zum Commodore 64«, Interface Age Verlag, oder »Das Commodore-64-Buch, Band 4. Ein Leitfaden für Systemprogrammierer«, Markt und Technik Verlag) und auch die Serie von Dr. Helmut Hauck: »Memory Map mit Wandervorschlägen«, die seit Ausgabe 11 / 1984 erscheint.

Ohne Hemmungen nutzen dürfen wir nur die Speicherstellen (jedenfalls beim C 64) $02 und $FB bis $FE.

Weil das doch recht mickrig ist, hat jeder Assembler-Programmierer spezielle Tipps, welche Zellen er noch mit welchen Vorsichtsmaßnahmen benutzt.

Wenn man bestimmte Routinen aus dem Betriebssystem oder dem Interpreter nicht aufruft, bleiben dazugehörige Zeropage-Adressen unbeeinflusst und sind dann für eigene Zwecke nutzbar.

Manchmal ist es notwendig, den alten Zustand einer Adresse nach Beendigung eigener Programme wieder herzustellen, manchmal nicht.

Interessant und viel beschrieben in allen möglichen Zeitschriften, Büchern etc. ist die Möglichkeit, die Notizen, die sich das Betriebssystem oder der Interpreter auf der Zeropage macht, zu verändern.

Im Prinzip schreibt man damit kleine Teile dieser Großprogramme um oder variiert Tabellenteile davon. Wie schon Dr. Hauck in seiner Serie sagt, geschieht das im Rahmen der »Tricks« mit irgendwelchen POKEs mehr oder weniger blind, weshalb auch Abstürze des Computers bevorzugt dabei festzustellen sind.

Warum Abstürze? Na, stellen Sie sich mal ein von Ihnen geschriebenes Programm vor - zum Beispiel das aus der letzten Ausgabe zur Berechnung der Summe einer arithmetischen Reihe - und POKEn Sie dann an Stelle irgendeines Befehlskodes, der dorthin gehört, jetzt eine 0 (also ein BRK) hinein. Die Wirkung dürfte ähnlich sein.

Wenn man allerdings die Funktion der betreffenden Speicherstelle genau kennt, lassen sich recht nützliche Änderungen hervorrufen, wie zum Beispiel die SchutzPOKEs für den BASIC-Speicher durch Verändern der Adressen $33, $34, $37 und $38.

Wir werden im Folgenden immer dann, wenn wir mit Zeropage-Adressierung arbeiten oder Routinen des Betriebssystems oder Interpreters untersuchen, spezielle Stellen der Nullseite kennen lernen.

Vorhin hatte ich noch angedeutet, dass man dann die Zeropage fast vollständig nutzen könne, wenn man auf den BASIC-Interpreter und das Betriebssystem verzichtet. Das ist tatsächlich möglich. Nur wird man dann erstaunt feststellen, wieviel Arbeit uns die computerinterne Software abnimmt oder anders herum: Viele bislang selbstverständliche Dinge werden wir dann plötzlich selbst programmieren müssen, und das kann ein hartes Brot sein!

Als Beispiel für ein Programm, das nicht nur die Zeropage-Adressierung verwendet, sondern sogar selbst komplett in der Zeropage steht, werden wir uns die CHRGET-Routine ansehen. Eine Klasse von Befehlen, die dort angewendet werden, die Vergleichsbefehle, soll zuvor noch gezeigt werden.

Die Vergleichsbefehle: CMP, CPX, CPY

Vergleichen heißt in englischer Sprache »to compare«, woraus Sie unschwer erkennen können, woher die Bezeichnung CMP und die CPs in CPX beziehungsweise CPY kommen. Verglichen wird jeweils der Akku-Inhalt (bei CMP), der Inhalt des X- (bei CPX) oder des Y-Registers (bei CPY) mit Daten, die der Compare-Befehl adressiert. Einige Beispiele werden Ihnen das klarer machen:

```
CMP #FF
```

vergleicht den Akku-Inhalt mit der Zahl $FF. Hier liegt die unmittelbare Adressierung vor, die ebenso für CPX und CPY verwendbar ist. Außerdem ist das dann ein 2-Byte-Befehl.

```
CPX 3000
```

vergleicht den Inhalt des X-Registers mit dem Inhalt der Speicherstelle $3000. Die absolute Adressierung ist also auch anwendbar (natürlich auch für CMP und CPY). Der Compare-Befehl besteht so aus 3 Bytes.

```
CPY A8
```

vergleicht den Inhalt des Y-Registers mit dem Inhalt der Zeropage-Stelle $A8.

Diese eben frisch kennen gelernte Zeropage-Adressierung ist bei allen drei Vergleichsbefehlen möglich und macht aus ihnen 2-Byte-Befehle. Für CPX und CPY sind das alle Möglichkeiten der Adressierung. CMP erlaubt weitere, die wir noch kennen lernen werden.

Nun interessiert uns natürlich noch, wie das Vergleichsergebnis zu erhalten ist! Bei diesen Befehlen geschieht Merkwürdiges: Die Vergleichsdaten werden vom Inhalt des Akkus (beziehungsweise X- oder Y-Registers) abgezogen, aber: Weder wird dieser Inhalt noch werden die adressierten Daten verändert!

Der Trick ist, dass drei Flaggen das Ergebnis anzeigen: die Negativ-Flagge N, die Null-Flagge Z und das Carry-Bit C. Diese Anzeige geschieht so:

1. Der Registerinhalt (Akku, X-, Y-Register) ist größer als die Ver-gleichsdaten. Dann ist das Carry-Bit = 1, die N-und die Z-Flagge = 0.
2. Der Registerinhalt ist gleich den Vergleichsdaten. Dann sind Carry- und Z-Flagge = 1, die N-Flagge ist = 0.
3. Der Registerinhalt ist kleiner als die Vergleichsdaten. Die N-Flagge ist dann = 1, Carry-und Zero-Flagge sind 0.

Damit Sie die Übersicht behalten können, wird das Ganze in Abbildung 10 als Schema gezeigt.

Sie werden sich vermutlich schon denken können, wie der Hase weiterläuft: Mit den Verzweigungsbefehlen prüfen wir die Flaggen und springen die gewünschten weiteren Programm-Routinen an.

FLAGGE	Akku, X, Y > DATEN	Akku, X, Y = DATEN	Akku, X, Y < DATEN
N	0 oder 1	0	0 oder 1
Z	0	1	0
C	1	1	0

Abb. 10: Flaggen bei den Vergleichsbefehlen

Die Kombination der Compare-Befehle mit den Verzweigungsoperationen wird Ihnen im weiteren Verlauf dieses Kurses noch geläufig werden. Ein Beispiel sehen Sie nachher ebenfalls in der CHRGET-Routine. Leider muss ich Sie immer noch etwas vertrösten, denn mit Verstand begreifen lässt sich diese Routine nur dann, wenn man etwas mehr über die Kodierung von Zeichen weiß. Deswegen werden wir uns nun noch mit dem ASCII-Kode und dem Commodore-ASCII herumschlagen.

Zeichenkodierung mit dem ASCII- und dem Commodore-ASCII-Kode

ASCII ist die Abkürzung von »American Standard Code for Information Interchange«, und das heißt auf deutsch »amerikanischer Standard-Kode zum Informationsaustausch«. Diese Zeichenverschlüsselungsart ist international als ISO-7-Bit-Kode genormt, und es wäre wirklich nett, wenn alle sich daran halten würden.

Tatsächlich aber finden wir zum Beispiel bei unserem C 64 eine Abart des Normkodes, den Commodore-ASCII-Kode. Über die damit erzwungenen Umrechnungen können alle diejenigen Dramen erzählen, die zum ersten Mal einen (Nicht-Commodore-)Drucker an ihr Gerät anschließen oder aber blauäugig in den Online-Betrieb mit anderen Computern eintreten wollten.

Sehen wir uns zunächst einmal den ASCII-Kode an. Es handelt sich um einen 7-Bit-Kode, das heißt 128 Zeichen können in nur 7 Bits untergebracht werden (0000 0000 bis 0111 1111). Das achte Bit dient bei manchen Operationen mit Computer-Peripherie als Paritätsbit. Bei dieser Gelegenheit soll auch gleich erklärt werden, was Parität in diesem Zusammenhang bedeutet:

Werden Daten übertragen, muss immer mit Übermittlungsfehlern gerechnet werden. Das Paritätsbit dient dazu, festzustellen, ob ein Byte korrekt angekommen ist. Bei der so genannten geraden Parität zählt man die Einser im Byte zusammen und setzt Bit 7 auf 1, wenn sich eine ungerade Zahl ergibt. Mit dem Paritätsbit haben wir dann eine gerade Zahl.

Ist die Quersumme des Bytes schon gerade, bleibt Bit 7 eine Null. Ebenso gut kann man die ungerade Parität verwenden, indem dann Bit 7 so gewählt wird, dass sich immer eine ungerade Zahl ergibt. Welche Art der Parität zur Anwendung kommt, ist Vereinbarungssache.

Nehmen wir mal an, es sei gerade Parität gefordert und ein Byte mit der Information 0001 0110 soll übermittelt werden. Die Quersumme ist 3, also ungerade. Das Paritätsbit muss auf 1 gesetzt werden.

Wir senden das Byte 1001 0110. Der Empfänger überprüft zunächst auf gerade Parität und verwendet dann nur die Bits 0 bis 6. Doppelfehler, die mittels des Paritätsverfahrens nicht festgestellt werden können, sind sehr selten.

Leider kann auf diese Weise nur bemerkt werden, dass ein Übertragungsfehler aufgetreten sein muss, aber nicht, welcher. Die Information muss dann neu angefordert werden.

Sehen wir uns nun den Commodore-ASCII-Kode an. Durch die Einbindung der Grafikzeichen brauchen wir mehr als die 128

Kombinationen. Commodore benutzt deswegen einen 8-Bit-Kode. Mit dem BASIC-Befehl CHR$(x) können Sie sich alle 256 Möglichkeiten ansehen. Erschwerend kommt aber noch hinzu, dass wir nicht nur einen Zeichensatz, sondern deren vier zur Verfügung haben, die durch den jeweiligen Schreibmodus ansprechbar sind (Klein-/Großschriftmodus, Großschriftmodus, beide Modi mit Reverse ON oder OFF). Im Zeichen-ROM liegen insgesamt 512 Muster abrufbereit. Zu diesen kommen beim CHR$-Befehl noch eine ganze Reihe von Steuerzeichen hinzu ... die Verwirrung ist perfekt!

Wir wollen an dieser Stelle keine Entwirrung vornehmen, sondern durchschlagen den Gordischen Knoten, indem wir nur die ersten 128 Zeichen mit den ASCII-Zeichen vergleichen. In den Tabellen 9 und 10 finden Sie unsere Gegenüberstellung.

Tab. 9: ASCII-Kode (jeweils oben) und Commodore-ASCII-Kode (jeweils unten);

msn = most significant nibble

lsn = least significant nibble

msn / lsn	$ / binär	0	1	2	3	4	5	6	7
	binär	0000	0001	0010	0011	0100	0101	0110	0111
$	binär								
0	0	000	NUL	DLE	SP	0	@	P		p
			NUL	DLE	SP	0	@	P	CHR$(96)	CHR$(112)
1	0	001	SOH	DC1	!	1	A	Q	a	q
			SOH	DC1	!	1	A	Q	CHR$(97)	CHR$(113)
2	0	010	STX	DC2	"	2	B	R	b	r
			STX	DC2	"	2	B	R	CHR$(98)	CHR$(114)
3	0	011	ETX	DC3	#	3	C	S	c	s
			ETX	DC3	#	3	C	S	CHR$(99)	CHR$(115)
4	0	100	EOT	DC4	$	4	D	T	d	t
			EOT	DC4	$	4	D	T	CHR$(100)	CHR$(116)
5	0	101	ENQ	NAK	%	5	E	U	e	u
			ENQ	NAK	%	5	E	U	CHR$(101)	CHR$(117)
6	0	110	ACK	SYN	&	6	F	V	f	v
			ACK	SYN	&	6	F	V	CHR$(102)	CHR$(118)
7	0	111	BEL	ETB	'	7	G	W	g	w
			BEL	ETB	'	7	G	W	CHR$(103)	CHR$(119)
8	1	000	BS	CAN	(	8	H	X	h	x
			BS	CAN	(	8	H	X	CHR$(104)	CHR$(120)
9	1	001	HT	EM	)	9	I	Y	i	y
			HT	EM	)	9	I	Y	CHR$(105)	CHR$(121)
A	1	010	LF	SUB	*	:	J	Z	j	z
			LF	SUB	*	:	J	Z	CHR$(106)	CHR$(122)
B	1	011	VT	ESC	+	;	K	[	k	{
			VT	ESC	+	;	K	[	CHR$(107)	CHR$(123)
C	1	100	FF	FS	,	<	L	\	l	¦
			FF	FS	,	<	L	£	CHR$(108)	CHR$(124)
D	1	101	CR	GS	-	=	M	]	m	}
			CR	GS	-	=	M	]	CHR$(109)	CHR$(125)
E	1	110	SO	RS	.	>	N	↑	n	~
			SO	RS	.	>	N	↑	CHR$(110)	CHR$(126)
F	1	111	SI	US	/	?	O	-	o	<DEL>
			SI	US	/	?	O	←	CHR$(111)	CHR$(127)

Einige Kombinationen dienen als Steuer-Kodes. (Die Bedeutung der dabei verwendeten Abkürzungen sehen Sie rechts.)

Nur ein Teil dieser Kodes wird tatsächlich genutzt. Andere haben - je nach Gerät, an das sie gesandt werden - unterschiedliche Bedeutungen. Denken Sie dabei nur mal an die verschiedenen Betriebssysteme des Commodore-Druckers 1526, wo man bei dem einen mit CHR$(1), bei dem anderen mit CHR$(14) den Breitschrift-Modus einschaltet. Innerhalb unseres Computers werden offensichtlich bestimmte Kodes anders genutzt. Das sind:

An Stelle von	geschieht Folgendes
ENQ	Zeichen weiß
BS	Blockieren der Umschaltung Klein-/Großschrift
HT	Zulassen der obigen Umschaltung
DC1	Cursor abwärts
DC2	Reverse-Modus an
DC3	Cursor in HOME-Position
DC4	INST/DEL
FS	Zeichen rot
GS	Cursor rechts
RS	Zeichen grün
US	Zeichen blau

Der auffälligste Unterschied ist der, dass beim Commodore-ASCII an Stelle der Kleinbuchstaben Grafikzeichen liegen. Sollte an Stelle des Normalmodus' der Klein- / Großschriftmodus eingeschaltet sein, findet man an Stelle der Großbuchstaben die kleinen.

Jetzt haben wir alle nötigen Kenntnisse, um die CHRGET-Routine in unserem Computer zu verstehen.

Die CHRGET-Routine

Das Kürzel CHRGET kommt von »Get a character«, was bei uns heißt: »Hole ein Zeichen«. Es handelt sich um eine sehr häufig benutzte Routine unseres BASIC-Interpreters, die - wie schon vorhin erwähnt - komplett in der Zeropage steht. Wenn Sie mit

NUL	NULl	
SOH	Start Of Heading	Beginn des Kopfs
STX	Start of TeXt	Textbeginn
ETX	End of TeXt	Textende
EOT	End Of Transmission	Übertragungsende
ENQ	ENQuiry	Anfrage
ACK	ACKnowledge	Bestätigung
BEL	BELl	Klingel
BS	BackSpace	Zurücksetzen
HT	Horizontal Tabulator	horizontaler Tabulator
LF	Line Feed	Zeilenvorschub
VT	Vertical Tabulator	vertikaler Tabulator
FF	Form Feed	Formatvorschub
CR	Carriage Return	Wagenrücklauf
SO	Shift Out	Rückschaltung
SI	Shift In	Dauerumschaltung
DLE	Data Link Escape	Datenverbindungsumschaltung
DC1-4	Device Control 1-4	Gerätesteuerung
NAK	Negative AcKnowledge	Negativ-Bestätigung
SYN	SYNchronous idle	Synchronisations-Leerlauf
ETB	End of Transmission Block	Ende des Übertragungsblocks
CAN	CANcel	annullieren
EM	End of Medium	Datenträgerende
SUB	SUBstitute	ersetzen
ESC	ESCape	Umschaltung
FS	File Separator	Dateitrennzeichen
GS	Group Separator	Gruppentrennzeichen
RS	Record Separator	Satztrennzeichen
US	Unit Separator	Einheitentrennzeichen
SP	SPace	Leerzeichen
DEL	DELete	Löschzeichen

Tab. 10: Bedeutung der Abkürzungen im ASCII-Kode

dem SMON mal nachsehen wollen, dann geben Sie den Befehl D 0073 008B ein. Sie haben dann die komplette Routine vor sich:

```
0073   E6 7A     INC 7A
0075   D0 02     BNE 0079
0077   E6 7B     INC 7B
0079   AD 2502   LDA 0225
007C   C9 3A     CMP #3A
007E   B0 0A     BCS 008A
0080   C9 20     CMP #20
0082   F0 EF     BEQ 0073
0084   38 SEC
0085   E9 30     SBC #30
0087   38 SEC
0088   E9 DO     SBC #D0
008A   60 RTS
```

Eventuell sieht die Zeile 0079 bei Ihnen anders aus. Das liegt dann an den Speicherstellen 7A und 7B, welche einen Zeiger darstellen (LSB = 7A und MSB = 7B), der bei Ihnen gerade auf einen anderen Platz zeigt als auf $0225. Diese CHRGET-Routine besteht aus drei Teilen:

Zeilen 0073 bis 0079
Weiterstellen des CHRGET-Zeigers und Einladen des dadurch angezeigten Speicherzelleninhaltes in den Akku.

Zeilen 007C bis 0082
Prüfroutinen

Zeilen 0084 bis 008A
Flaggen-Routinen

Im ersten Teil haben wir schon gleich etwas Neues vor uns: ein sich selbst veränderndes Programm. Die Speicherstelle (aus dem BASIC-Eingabepuffer), aus der der Akku ein Zeichen holt, wird mit INC 7A um 1 weitergezählt.

Dabei handelt es sich um das LSB der Adresse, und die nächste Zeile prüft, ob ein Überlauf (255 + 1) stattgefunden hat: BNE 0079. Diese Technik kennen wir schon aus den letzten Folgen: Bei Überlauf wird die Z-Flagge auf 1 gesetzt und der BNE-Befehl führt keinen Sprung herbei. Den Offset von 02 können wir leicht nachrechnen: Der Programmzähler steht schon auf 0077. Die Zieladresse 0079 ist also noch 2 Bytes entfernt. Hat eine Überschreitung des Höchstwertes 255 stattgefunden, muss das dazugehörige MSB um 1 erhöht werden. Dies erledigt die nächste Zeile: INC 7B

In beiden Fällen ist nun der Zeiger 7A/7B um eine Stelle weitergerückt, und der Inhalt der dadurch angezeigten Speicherstelle wird in den Akku geladen. Zwei Dinge können wir uns aus diesem kurzen Programmteil merken:

1. Wie man eine 16-Bit-Zahl hoch- (oder auch herunter-)zählt
2. eine Möglichkeit, Zeiger einzusetzen

Wir werden noch eine Reihe anderer Zeigertypen kennen lernen und sehen, dass es nicht immer so direkt zugeht wie hier.

Im zweiten Teil finden wir die Prüfroutinen. Die Vergleichsbefehle beschränken sich auf den Akkuinhalt, also CMP.

CMP #3A testet, in welcher Beziehung das im Akku befindliche Zeichen zum Wert $3A = dezimal 58 steht. Erinnern wir uns an das Schema in Abbildung 10:

1. Commodore-ASCII-Kode im Akku größer als 58, also Zeichen hinter dem Doppelpunkt (Buchstaben, Grafikzeichen, einige Sonderzeichen). Dann ist die Carry-Flagge = 1, N- und Z-Flagge sind 0.
2. Im Akku steht genau der Kode 58, also der Doppelpunkt. Dann sind Carry-Bit und Z-Flagge = 1, nur die N-Flagge ist = 0.
3. Der Kode des Zeichens im Akku ist kleiner als 58 (das wären alle Zahlen, einige Sonderzeichen und Steuerzeichen). In diesem Fall ist die N-Flagge = 1. Die beiden anderen Flaggen zeigen Null.

Der nun folgende Befehl BCS 008A überprüft die Carry-Flagge. Wenn sie gesetzt ist, wenn also der Kode im Akku größer oder gleich dem eines Doppelpunktes (58) ist, springt der Programmzähler zum RTS. Der Kode (und auch die Flaggen) wird unverändert zum aufrufenden Hauptprogramm weitergegeben.

Zur Übung können Sie ja noch mal den Offset nachrechnen. Der Rest des Programms wird nur noch durchlaufen, wenn Kodes kleiner als 58 im Akku stehen.

Die nächste Zeile (CMP #20) dient zum Vergleich des Space-Codes $20 = dezimal 32 (Leertaste). Die Flaggen treten dann, wie schon oben beim ersten Vergleich gezeigt, je nach Akku-Inhalt auf.

Durch die Verzweigung BEQ 0073 erfolgt dann ein Rücksprung zum Beginn der CHRGET-Routine, wenn die Z-Flagge gesetzt ist, also ein Space-Kode im Akku liegt. Somit werden die Leerzeichen einfach übersprungen und das nächste Zeichen geholt.

Alle anderen Zeichen, die bis hierher durchgehalten haben, werden nun im letzten Teil der CHRGET-Routine einer Prozedur unterworfen, die ich Flaggen-Routine genannt habe.

Durch zwei aufeinander folgende Subtraktionen, die insgesamt den Wert im Akku unverändert lassen (es wird 256 abgezogen), wird die Carry-Flagge beeinflusst. Verfolgen wir, was da passiert:

SEC dient als Vorbereitung für die folgende Subtraktion. SBC #30 zieht vom Akku-Inhalt $30 = dezimal 48 ab.

Wir wissen inzwischen, dass das der Addition des Zweierkomplements entspricht. Dieses ist (rechnen Sie mal nach!) 1101 0000. Nehmen wir mal an, wir hätten den Kode der Zahl 4 (also dezimal 52 oder $34) im Akku stehen. Die Rechnung sieht dann so aus:

```
52          0011 0100
 +          1101 0000
 4      (1) 0000 0100
```

Das Ergebnis ist also 4, der Übertrag wird vernachlässigt.

Als anderes Beispiel sei nun der Kode für das Ausrufungszeichen im Akku (dezimal 33 = $21 = binär 0010 0001). Die Rechnung ist dann:

```
 33         0010 0001
  +         1101 0000
-15         1111 0001
```

Das Ergebnis ist -15. Alle Kodes, die nicht für Zahlen stehen, haben nach dieser Subtraktion ein negatives Ergebnis im Akku hinterlassen und durch das »Borgen« das Carry-Bit gelöscht. Nun machen wir weiter ab Zeile 0087:

```
SEC
SBC #D0
```

Wir ziehen $D0 = dezimal 208 ab. Das Zweierkomplement ist: ... Doch da kommen wir ins Stocken! Denn dieses Zweierkomplement ist nicht mehr mit 8-Bit-Zahlen darzustellen. Schon die Zahl 208 im Binärformat (1101 0000) würde als negative Zahl angesehen werden, weil Bit 7 gleich 1 ist.

Wir machen es uns einfach und sagen, dass sich das Zweierkomplement wie bisher bilden lässt, aber dabei das Carry-Bit mit einbezogen wird.

Unser Zweierkomplement ist dann also 0011 0000 und das Carry-Bit ist gelöscht. Nun nehmen wir unser erstes Beispiel. Dort war nach der Subtraktion im Akku eine 4 verblieben:

```
            0000 0100
+           0011 0000
            0011 0100
```

Das ist wieder unser ursprünglicher Wert dezimal 52 = $34 = Kode für die Zahl 4. Das Carry-Bit bleibt gelöscht.

Im zweiten Beispiel mit dem Ausrufungszeichen stand noch im Akku eine -15:

```
          1111 0001
+         0011 0000
(1)       0010 0001
```

Da haben wir wieder den Kode für das Ausrufungszeichen ($21 = dezimal 33) im Akku und ein gesetztes Carry-Bit. Was kommt also bei der CHRGET-Routine heraus?

1. Alle Zeichen außer dem Space werden unverändert an das aufrufende Programm über den Akku weitergegeben. Space wird unterdrückt.
2. Bei allen Zeichen außer bei den Zahlen ist das Carry-Bit gesetzt.
3. Manche der aufrufenden Routinen überprüfen außer dem Zustand der Carry-Flagge auch den der Z- oder N-Flagge, die ja beim ersten CMP-Befehl ebenfalls gesetzt werden. So liefert die CHRGET-Routine noch weitere Informationen.

In der einschlägigen Literatur stoßen Sie auch auf eine Routine, die CHRGOT genannt wird. Es handelt sich dabei ebenfalls um die hier beschriebene CHRGET-Routine, nur erfolgt der Einsprung nicht bei $0073, sondern bei $0079.

Der Zeiger $007A / 7B wird in diesem Fall nicht weitergestellt. Das vorher schon einmal in den Akku geladene Zeichen wird damit noch einmal angesprochen (»got« ist die Vergangenheitsform von get).

Mit dem CHRGET-Programm haben wir eines der wichtigsten Unterprogramme unserer computerinternen Software kennen gelernt. Will man sich Interpreter-Routinen zu Nutze machen, stolpert man ständig darüber. Außerdem liegt die CHRGET-Routine im RAM. Das bedeutet, dass wir sie ohne Weiteres für unsere Zwecke verändern können.

Ein Beispiel für so eine Änderung hat Christoph Sauer in seiner Serie über den »gläsernen VC 20« in der 64'er 9 / 1984 (Seite 158) gezeigt. Dort wird die CHRGET-Routine nach dem LDA angezapft und auf das Pi-Zeichen geprüft, das neuen Befehlen vorangestellt wurde. Sehen Sie sich das Programm dort (auf Seite 160 f.) mal genau an, denn man kann viel durch Nachvollziehen fremder Programme für die eigene Programmiertechnik lernen.

Wir werden im Verlauf dieser Serie noch andere Möglichkeiten behandeln, die CHRGET-Routine zu verändern.

Damit sei es für diesmal genug. Als Assembler-Alchemisten gehören Sie jetzt zu den fortgeschrittenen Eleven, denn Sie können immerhin schon so trickreiche Programme wie die CHRGET-Routine nachvollziehen.

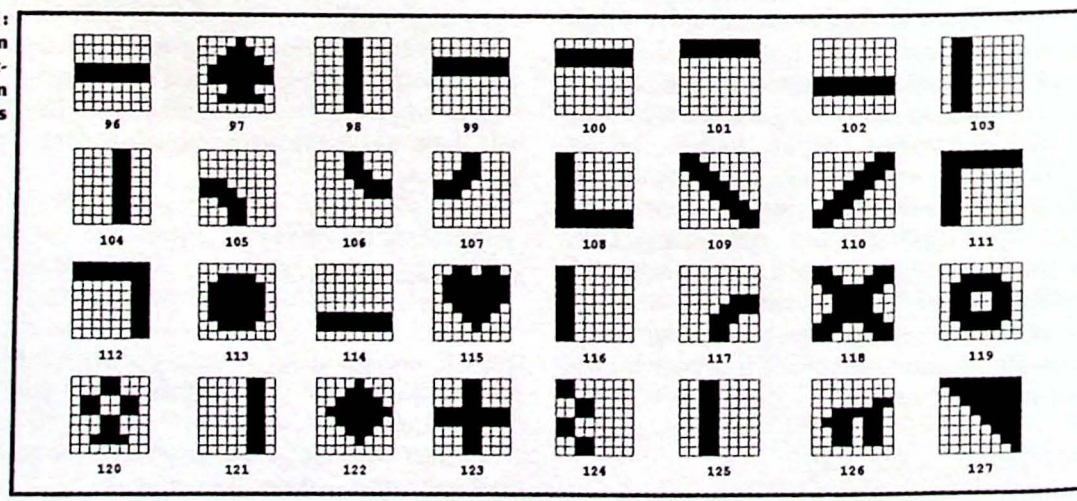


Abb. 11: Grafikzeichen zu den entsprechenden CHR$-Kodes

Assembler ist keine Alchemie
Folge 6

Erschienen in: 64'er 2/1985
Verantwortlicher Redakteur: Georg Klinge
Autor: Heimo Ponnath

In der vorangegangenen Ausgabe haben wir die relative und die Zeropage-Adressierung kennen gelernt. Heute kommt die indizierte Adressierung dran, und natürlich sehen wir uns wieder einige neue Assembler-Befehle an.

Wir werden uns einige Gedanken machen über die so genannten Fließkommazahlen und den BASIC-Befehl USR. Auch die Speicherorganisation unseres Computers soll uns noch mal beschäftigen.

Zunächst die indizierte Adressierung. Indizieren heißt, etwas mit einem Index, also einem Zeichen oder einer Nummer, zu versehen. Beispielsweise bezeichnet man in der Mathematik die beiden Lösungen einer quadratischen Gleichung häufig als X1 und X2. Dabei ist dann die Ziffer (1 oder 2) der Index und X ist eine indizierte Größe. Man geht also von einer festgelegten Grundmenge (Lösungsmenge X) aus und trifft durch den Index eine weitere Unterscheidung.

So ähnlich können wir uns auch die Funktion der indizierten Adressierung bei der Assembler-Programmierung vorstellen.

Nehmen wir als Beispiel den Befehl `LDA 1500,X`. Man spricht hier von einer absolut-X-indizierten Adressierung.

Das Assemblerwort `LDA` ist uns bekannt: Lade den Akku. Woher soll der für den Akku bestimmte Inhalt geholt werden? Aus der Speicherzelle, die sich durch 1500 plus Inhalt des X-Registers ergibt. Steht also im X-Register zum Zeitpunkt des Befehlsaufrufs eine 5, dann wird der Akku aus Speicherzelle 1500 + 5, also 1505, geladen. Das X-Register kann Werte von 0 bis $FF (dez. 255) enthalten.

Die Ähnlichkeit sieht also so aus: Aus einer Gesamtmenge von 256 Adressen, die durch die Anfangsadresse (bei unserem Beispiel 1500) und die möglichen 256 Belegungen des X-Registers festgelegt sind (die Grundmenge), werden je nach X-Registerinhalt einzelne Adressen unterschieden und adressiert. Das X-Register fungiert dabei als ein Index, weswegen man in der Literatur auch oft die Bezeichnung »Indexregister X« findet.

Ebenfalls als Index-Register kann das Y-Register dienen, was zum Beispiel zum Befehl `LDX 1500,Y` führen kann. Dies ist dann eine absolut-Y-indizierte Adressierung.

Genauso, wie man die normale absolute Adresse (also zum Beispiel 1500) als Basis der Indizierung durch das X- oder das Y-Register verwenden kann, ist das auch mit eine Zeropage-Adresse möglich. So gibt es zum Beispiel die Befehle `LDY 2B,X` oder `STX 19,Y`.

Man nennt diese Art der Adressierung dann Zeropage-absolut-X-indiziert beziehungsweise -Y-indiziert.

Weil die Zeropage aber nur 256 Adressen umfasst, andererseits jedoch die Indexregister auch 256 Werte annehmen können, kann es geschehen (wenn man nicht aufpasst), dass die Summe aus der Basisadresse (zum Beispiel $2B) und dem Indexregisterinhalt größer als 256 wird. Wenn zum Beispiel in

Tab. 11: Anwendbarkeit der indizierten Adressierungsarten auf die bisher gelernten Assembler-Befehle

Befehl	indizierte Adressierung			
	absolut		Nullseite absolut	
	X	Y	X	Y
Folge 2				
LDA	+	+	+	-
LDX	-	+	-	+
LDY	+	-	+	-
STA	+	+	+	-
STX	-	-	-	+
STY	-	-	+	-
RTS	/	/	/	/
Folge 3				
INX	/	/	/	/
INY	/	/	/	/
INC	+	-	+	-
DEX	/	/	/	/
DEY	/	/	/	/
DEC	+	-	+	-
SED	/	/	/	/
CLD	/	/	/	/
BNE	/	/	/	/
Folge 4				
ADC	+	+	+	-
CLC	/	/	/	/
SBC	+	+	+	-
SEC	/	/	/	/
BEQ	/	/	/	/
BCC	/	/	/	/
BCS	/	/	/	/
BMI	/	/	/	/
BPL	/	/	/	/
BVC	/	/	/	/
BVS	/	/	/	/
Folge 5				
CMP	+	+	+	-
CPX	-	-	-	-
CPY	-	-	-	-
Folge 6				
BIT	-	-	-	-
CLV	/	/	/	/
NOP	/	/	/	/
TAX	/	/	/	/
TAY	/	/	/	/
TXA	/	/	/	/
TYA	/	/	/	/
JMP	-	-	-	-
JSR	-	-	-	-

dem Befehl LDA FE,X der X-Registerinhalt 2 beträgt, ergäbe sich $FE + $02 = $0100. In diesem Fall wird aber nicht der Inhalt von $0100 in den Akku geladen, sondern der Befehl spricht die Speicherstelle $00 an.

Der Grund dafür liegt in der Tatsache, dass unser Prozessor den Befehl als 2-Byte-Befehl interpretiert - das 2. Byte ist die Zeropageadresse, die sich als Summe ergibt - und deswegen nur das LSB der Adresse beachtet. Von $0100 ist das LSB aber $00. Mit anderen Worten: Die Zeropage-absolut-indizierten Befehle lassen nur einen Zugriff auf die Zeropage selbst zu. Dieses Verhalten muss man beim Programmieren beachten.

Wir wollen noch mal zusammenfassen: Vier neue Adressierungsarten haben wir kennen gelernt: absolut-X-indiziert zum Beispiel LDA 1500,X; absolut-Y-indiziert zum Beispiel LDA 1500,Y; Zeropage-absolut-X-indiziert zum Beispiel LDA 2B,X; Zeropage-absolut-Y-indiziert zum Beispiel LDX 2B,Y.

Die Verwendung des Y-Registers als Indexregister ist stark eingeschränkt. Nur bei wenigen Befehlen ist sie erlaubt (tatsächlich nur LDX und STX bei Zeropage-absolut-indizierter Adressierung).

In Tabelle 11 sehen Sie, welche bisher behandelten Befehle auf welche Weise mit der indizierten Adressierung verwendet werden dürfen.

Es gibt noch zwei weitere Arten einer indizierten Adressierung, auf die wir noch zu sprechen kommen werden.

Einige Nachzügler: Die Befehle BIT, CLV, NOP und TAX, TAY, TXA, TYA

Wir wollen noch ein bisschen aufräumen: Ein paar Befehle, die bisher zu keinem Gebiet so richtig passten, sollen jetzt behandelt werden.

```
+   anwendbar
-   nicht erlaubt
/   weder absolute noch Zeropage-
    Adressierung möglich
```

BIT

Dieser Befehl heißt »Bit-Test« und passt von daher eigentlich zu den in der letzten Ausgabe behandelten Vergleichsbefehlen. Die Behandlung der Flaggen ist aber völlig anders. Nehmen wir das Beispiel `BIT 1500`.
Folgendes passiert: Der Inhalt der Speicherstelle $1500 wird mit dem Inhalt des Akkus UND-verknüpft, das Ergebnis in der Z-Flagge angezeigt und Bit 7 sowie Bit 6 von $1500 in die N- beziehungsweise die V-Flagge übertragen. Weder der Akku noch der Inhalt von $1500 verändern sich dabei.

Das ging ein bisschen holterdipolter. Sehen wir uns das jetzt mal ganz langsam Schritt für Schritt an! Zunächst die UND-Verknüpfung: Bit für Bit wird der Akku-Inhalt mit dem Inhalt der adressierten Speicherstelle UND-verknüpft. Dabei gelten folgende Regeln:

0 UND 0 = 0
0 UND 1 = 0
1 UND 0 = 0
1 UND 1 = 1

Nur dann also, wenn die entsprechenden Bits im Akku und in 1500 gleich 1 sind, ergibt sich bei der UND-Verknüpfung eine 1. Man stellt so etwas meist in einer so genannten Wahrheitstabelle zusammen (Tabelle 12).

Nehmen wir als Beispiel mal an, im Akku stünde $0A und in der Speicherstelle $1500 wäre $09 enthalten. Die UND-Verknüpfung sieht dann so aus:

```
Akku  $0A  0000 1010
1500  $09  0000 1001
UND        ---------
           0000 1000
```

Das Ergebnis ist also $08. In der Z-Flagge wird in dem Fall, dass das Ergebnis der UND-Verknüpfung ungleich Null ist (wie hier) eine Null angezeigt, sonst eine 1. Wir haben in unserem Zahlenbeispiel mit dem `BIT`-Befehl überprüft, ob die Bits 1 und 3 in Speicherstelle $1500 gelöscht sind. Dazu haben wir in den Akku eine so genannte Maske (hier also $0A) geladen. Das Ergebnis sagt uns, dass nicht beide Bits gelöscht waren. Wäre der Inhalt von $1500 beispielsweise $10 gewesen (`0001 0010`), hätten wir in der Z-Flagge eine 1 gefunden.

UND	0	1
0	0	0
1	0	1

Tab. 12: Wahrheitstabelle der logischen Verknüpfung UND

Daher der Name »Bit-Test«: Durch geeignete Maskenwahl kann praktisch jedes Bit überprüft werden. Dabei werden weder der Akku-Inhalt noch der Inhalt der angesprochenen Speicherstelle verändert.

Der `BIT`-Befehl hat aber noch mehr Auswirkungen: Die Bits 6 und 7 der geprüften Speicherzelle findet man nach Befehlsausführung in zwei Flaggen noch mal:

- Bit 7 in der N-Flagge
- Bit 6 in der V-Flagge

Damit kann man beispielsweise überprüfen, ob sich am adressierten Ort eine negative Zahl befindet. Alle drei Flaggen können ja nun mit den Branch-Befehlen abgefragt werden. Sie erkennen sicherlich schon, wie vielseitig dieser merkwürdige `BIT`-Befehl einsetzbar ist.

Adressierbar ist `BIT` entweder absolut (wie im obigen Beispiel) oder Zeropage-absolut. Je nachdem liegt er dann als 3-Byte- oder als 2-Byte-Befehl vor.

CLV

Dieser Befehl heißt »CLear oVerflow flag«, also »lösche die Überlauf-Flagge«. Die V-Flagge war - wie Sie sich erinnern werden - unsere rote Ampel bei Rechenoperationen (siehe Folge 4).
Es ist ein 1-Byte-Befehl mit impliziter Adressierung.

Interessant daran ist, dass es keinen Befehl gibt, der das Gegenteil - also das Setzen der V-Flagge - bewirkt.

NOP

NOP steht für »No OPeration«, was bedeutet »keine Tätigkeit«. Das ist der Nichtstu-Befehl. Er tut aber doch etwas: Er sorgt dafür, dass der Befehlszähler weitergezählt wird, und bewirkt eine Verzögerung von 2 Taktzyklen.

NOP ist ein 1-Byte-Befehl mit impliziter Adressierung. Er wird in fertigen Programmen nur selten verwendet: zur Erzeugung einer kurzen definierten Verzögerung. Meist gebraucht man ihn bei der Erstellung eines Programms als Platzhalter oder bei der Fehlersuche, um zum Beispiel unerwünschte Sprünge zu ersetzen. Die Transporteure sind: TAX, TAY, TXA und TYA.

Ab und zu ist es nötig, Registerinhalte untereinander auszutauschen. Viele Dinge (Addition, Subtraktion und so weiter) können nur im Akku geschehen. Wenn wir eine solche Operation beispielsweise mit dem Inhalt des X-Registers ausführen wollen, verschieben wir diesen Inhalt mit dem Befehl TXA. Das bedeutet »Transfer X into Accumulator«, also »übertrage X-Register in den Akku«. Analog verwendet man TYA, um Y-Register-Inhalte in den Akku zu schieben oder für den umgekehrten Weg TAY beziehungsweise TAX (Akkuinhalt ins Y- beziehungsweise ins X-Register schieben). Genau genommen wird nicht übertragen, sondern nur kopiert: Die Register, aus denen verschoben wird, bleiben unverändert.

Weil die jeweiligen Zielorte der Verschiebung (Akku, X- oder Y-Register) vom neuen Inhalt überschrieben werden, können sich auch Flaggen ändern. Betroffen sind von dieser Möglichkeit die N- und die Z-Flagge. Alle vier Befehle bestehen aus einem Byte und können natürlich nur implizit adressiert werden.

So springen die Assembler-Alchemisten: JMP, JSR

JMP und JSR entsprechen ungefähr den vom BASIC her bekannten Befehlen GOTO und GOSUB. JMP kommt von »JuMP to address«, also »springe zur angegebenen Adresse«. Nehmen wir uns wieder ein Beispiel vor: JMP 1500 bewirkt einen Sprung zur Adresse 1500.

Das funktioniert so: In den Programmzähler werden LSB und MSB der Zieladresse geladen. Das war dann auch schon der Sprung, denn der Programmzähler ist der Pfadfinder des Computers: Die Adresse, die dort steht, wird als Nächste bearbeitet.

Schalten Sie doch mal den SMON ein (oder einen anderen Monitor) und sehen Sie sich das mit folgenden Befehlen an:

```
1400 JMP 1500
```

Dort unterbrechen wir den Computer mit

```
1500 BRK
```

So weit, so gut: Wir starten mit dem SMON-Kommando G 1400 und erhalten eine Registeranzeige mit dem Programmzählerstand 1501.

Genau das hatten wir ja erwartet. Weniger durchschaubar ist das folgende Beispiel:

```
1400 LDA #00
1402 LDX #16
1404 STA 1300
1407 STX 1301
140A JMP (1300)
```

Dazu gehört dann noch die Programmzeile

```
1600 BRK
```

Wenn Sie das genauso eingegeben haben und dann mittels G 1400 starten, erhalten Sie eine Registeranzeige mit dem Programmzählerstand 1601.

Schon an der neuen Schreibweise des Arguments in Zeile 140A werden Sie bemerkt haben, dass hier nicht mehr die normale absolute Adressierung wie zuvor angewendet wird. Dies ist eine neue Form: Die indirekte Adressierung. Indirekt deswegen, weil wir nicht mehr direkt die Zieladresse angeben,

sondern einen so genannten Vektor. Ein Vektor besteht aus zwei aufeinander folgenden Speicherzellen (hier also 1300 und 1301), die in der Form LSB / MSB die eigentliche Zieladresse enthalten.

Das LSB von $1600 ist $00. Das haben wir über den Akku nach $1300 geladen. Das MSB $16 kam durch das X-Register an seinen Platz $1301:

```
Zieladresse   16     00
              MSB    LSB
               ↑      ↑
Vektor        1301   1300
```

Das ist die Methode der toten Briefkästen, die in Kreisen der Assembler-Alchemisten anscheinend genauso beliebt ist wie bei Agenten. So wie diese im hohlen Baum die Treffpunktanschrift hinterlegt finden, verlässt sich unser Computer auf die Speicherstellen 1300 und 1301 für die Angabe der Zieladresse.

Diese Art der Adressierung ist im wahrsten Sinn des Wortes ein Unikum: Es gibt sie nämlich nur für den JMP-Befehl! Davon wird allerdings dann auch recht häufig Gebrauch gemacht, zum Beispiel im Betriebssystem unseres Computers.

Aber darüber und über die Vektoren, die dazu verwendet werden, soll ein andermal berichtet werden.

Wir dürfen nämlich nicht den anderen Sprungbefehl JSR vergessen. JSR steht für »Jump to SubRoutine«, was eingedeutscht etwa »springe zum Unterprogramm« bedeutet.

Genauso wie in BASIC Unterprogramme durch »GOSUB Zeilennummer« aufgerufen werden, kann das auch hier geschehen: durch JSR Adresse. Hier ist nur die absolute Adressierung möglich. Das erste Beispiel soll uns zeigen, wie dieser Befehl funktioniert:

```
                    1400 JSR 1500
Dort steht dann erst mal 1500 BRK
```

Noch nicht starten! Zunächst einmal verzeihen Sie mir die Programmierer-Todsünde, aus einem Unterprogramm heraus den Programmablauf zu beenden. Ich werd's auch nie wieder tun. Hier geschieht das nur zu Lehrzwecken.

Was läuft ab? Der Programmzählerinhalt plus 2 wird auf den Stapel gelegt und dann die Adressse 1500 in den Programmzähler geladen.

Ebenso kurz wie unklar! Was ist denn ein Stapel? Also langsam, Schritt für Schritt.

Der Sinn von Unterprogrammen ist ja, dass der Computer nach Ende der Bearbeitung wieder ins aufrufende Hauptprogramm zurückkehrt. Er muss sich aber dazu irgendwo merken, von wo aus er zum Unterprogramm gesprungen ist.

Dazu verwendet er den Stapel. Das ist ein Speicherbereich ($0100 bis $01FF), der direkt vom Prozessor aus verwaltet wird. Die genaue Architektur und Handhabung dieses »Prozessor Stack« werden wir noch in einer späteren Folge kennen lernen. Uns soll hier nur interessieren, dass es einen Zeiger gibt, der auf den nächsten freien Platz im Stapel weist, und dass dieser Speicher von oben nach unten gefüllt wird (wie in BASIC bei den Strings).

Wenn Sie mithilfe des SMON mal in den Stapel hineinsehen wollen, dann geben Sie doch mal ein M 0100 01FF.

Was nun genau bei Ihnen drin steht, ist sehr von der vorherigen Nutzung Ihres Computers abhängig. Der Mikroprozessor nutzt den Stapel bei sehr vielen Tätigkeiten. Es kommt auch nur auf den Teil des Stapels an, der durch den Stapelzeiger als gefüllt bezeichnet wird. Der Stapelzeiger wird beim SMON in der Registeranzeige als SP angezeigt. Wenn Ihr Stapelzeiger (prüfen Sie das doch mal durch Eingabe von R) nun zum Beispiel F6 zeigt, dann bedeutet das, dass alle Stapelplätze von $01F6 an abwärts frei und die oberhalb bis $01FF besetzt sind. Beim Nachsehen mit M 01F0 01FF finden Sie dann beispielsweise:

```
:01F0 20 00 20 AA C1 FA C0 46
:01F8 E1 E9 A7 A7 79 A6 9C E3
```

Die Speicherstelle, auf die der Stapelzeiger weist, ist unterstrichen. Nun starten wir mit G 1400 unser kleines verbotenes Testprogramm. Es meldet sich die Registeranzeige.

Im Stapelzeiger steht jetzt F4 (oder eben Ihr vorangegangener SP minus 2). Wenn wir nun wieder mit M 01F0 01FF im Stapel nachsehen, dann finden wir im Gegensatz zur obigen Anzeige nun:

```
:01F0 20 AA C1 FA C0 02 14 46
                  ...  ↑  ↑
:01F8 E1 E9 A7 A7 79 A6 9C E3
```

Unterstrichen ist wieder das Ziel des Stapelzeigers, der jetzt zwei Plätze weitergerückt ist, um der durch Pfeile gekennzeichneten Adresse 1402 (als LSB / MSB) Platz zu machen. $1402 ist das letzte Byte des TSR-Befehls. Wie wir den Programmzähler kennen, ist er im Allgemeinen immer einen Schritt voraus.

Hier liegt er aber einen zurück, falls er nach Beendigung des Unterprogramms an der notierten Adresse weitermacht. Dazu kommen wir gleich noch. Was wir am Programmzähler aber auch noch nach Ablauf unseres kurzen Beispielprogramms ablesen können, ist die Tatsache, dass die Sprungadresse 1500 in ihn geschrieben wird und der Sprung somit stattgefunden hat.

Nun bauen wir das kleine Prográmmchen etwas um:

```
1400 JSR 1500
1403 BRK
```

Das Unterprogramm soll nur aus dem Rücksprung bestehen:

```
1500 RTS
```

Verlangen Sie nun noch vor dem Start eine Registeranzeige mit R und merken Sie sich den Wert des Stapelzeigers. Dann starten Sie das Programm mit G 1400 und achten Sie auf die neue Registeranzeige. Zwei Dinge interessieren uns:

1. Der Wert des Stapelzeigers ist unverändert geblieben.
2. Der Programmzähler weist nun auf $1404.

Wenn Sie nun noch mal mit dem M-Befehl des SMON in den Stapel sehen, werden Sie unter Umständen zwar noch die Adresse 1402 dort finden (dann nämlich, wenn wir den Stapel seit dem letzten Programm nicht verändert haben). Wie Sie aber inzwischen wissen, hätte durch den neuen JSR-Befehl noch mal 1402 dort eingetragen sein müssen. Das stand da auch einige Mikrosekunden lang ... bis der RTS-Befehl wirksam wurde. RTS macht ziemlich viel:

1. RTS holt die auf dem Stapel gespeicherte Adresse ab und schreibt sie in den Programmzähler.
2. RTS vermindert dabei den Stapelzeiger um 2.
3. RTS addiert zum Programmzähler eine 1.

Deswegen kann das Programm also bei $1403 weiterlaufen und der Programmzähler nun hinter dem BRK-Befehl stehen.

Machen Sie doch mal etwas anscheinend total Verrücktes: Starten Sie mit G 1500. Es gibt da zwei Möglichkeiten, was geschehen kann: Entweder stand da noch vom ersten unterbrochenen Testprogramm die Adresse 1402. Dann endete nun alles mit einer Registeranzeige, bei der der Stapelzeiger um 2 höher gerutscht ist.

Oder da stand diese Adresse nicht mehr. Dann befinden Sie sich nun wieder im BASIC. Wieso eigentlich? Als nächste Adresse finden Sie auf dem Stapel $E146 (dez. 57670). Diese Adresse + 1 wird ja durch RTS in den Programmzähler gerufen. Ein Sprung an diese Adresse ist ein Sprung in ein Programm des Betriebssystems. Haben Sie ein

ROM-Listing? Dann sehen Sie mal nach: Dort steht der Befehl ... RTS. Dieses neuerliche RTS holt nun die nächste Adresse vom Stapel: $A7E9 (dez. 42985). Diese Adresse + 1 im Programmzähler führt unseren Computer in die BASIC-Interpreter-Schleife, also ins BASIC zurück.

Wir haben so viel über den Stapel gehört, dass wir JSR fast schon wieder aus den Augen verloren haben. Deswegen noch mal eine kurze Übersicht:

a) JSR speichert den Programmzählerwert des letzten Bytes des Befehls auf dem Stapel, zum Beispiel 1402.
b) stellt dabei den Stapelzähler um 2 zurück, zum Beispiel von $F6 nach $F4.
c) schreibt in den Programmzähler die angegebene Zieladresse, zum Beispiel 1500.
d) Das Unterprogramm wird abgearbeitet, bis der RTS-Befehl auftaucht.
e) Dann wird die gemerkte Adresse + 1 in den Programmzähler geschrieben, zum Beispiel 1402 + 1 = 1403
f) und dabei der Stapelzähler wieder um 2 erhöht, zum Beispiel von $F4 wieder zu $F6.
g) Das Programm läuft nach dem JSR-Befehl weiter, zum Beispiel bei 1403.

Nun sollte eigentlich auch klar sein, warum ein Aussprung aus einem Unterprogramm oder ein Abbruch im Unterprogramm eine Programmierer-Todsünde ist: Der Stapelzeiger wird nicht zurückgestellt. Die gemerkte Rücksprungadresse versauert allmählich auf dem Stapel.

Noch schlimmer sind solche Sachen in einer Schleife, wo mehrfach aus dem Unterprogramm ausgebrochen wird: Hier ist der Stapel bald voll Müll, und der Computer beendet seine Zusammenarbeit mit dem Programmierer. Weil aber BASIC-Programme nichts anderes sind als eine Folge von Maschinenprogrammen, die je nach Befehl durch den Interpreter aneinandergereiht werden, ist das auch in BASIC eine Todsünde. Wir wollen aber nicht so hart mit uns umgehen: Wenn wir gelernt haben, wie man mit speziellen Assembler-Befehlen im Stapel herumschaufeln kann, dann haben wir bei richtiger Anwendung von vornherein jedenfalls in diesem Punkt die Absolution erhalten.

Alles fließt: Fließkommazahlen

Jeder, der tiefer in die Geheimnisse der Assembler-Alchemie eindringen will, muss sich vertraut machen mit der häufigsten Art der Zahlenverarbeitung in unserem Computer.

Das ist die Handhabung von Fließkommazahlen (auch Gleitkommazahlen genannt). Wir werden dazu folgende Fragen zu klären haben:

1. Was sind Fließkommazahlen?
2. Wie sehen sie im binären Zahlensystem aus?
3. Wie behandelt unser Computer positive und negative Fließkommazahlen?
4. Wie können wir als Programmierer Einfluss auf die Verarbeitung dieser Zahlen im Computer nehmen?

Die Behandlung dieser vier Fragen wird uns eine ganze Weile beschäftigen. Fangen wir mit der ersten an:

In Standardwerken der Mathematik werden Sie lange suchen müssen, um den Begriff »Fließkommazahl« zu finden. Im deutschen Sprachraum gibt es häufiger die Bezeichnung »wissenschaftliche Zahlendarstellung«. Das klingt sehr hochgestochen und ist eigentlich ganz einfach.

Die Zahl 1000 kann man auf verschiedene Weise darstellen:

1000 = 10 * 10 * 10 = 10^3 etc.

Die hochgestellte Zahl (in Computerschreibweise: Die Zahl hinter dem Hochpfeil) ist hier gleich der Anzahl der Stellen minus 1 (1000 hat vier Stellen, also ist die Hochzahl eine 3). Diese Hochzahl nennt man Exponent (vom lateinischen exponere = anzeigen,

herausheben). Nehmen wir nun einige andere Zahlen:

200 = 2 * 100 = 2 * 10^2
2500 = 2,5 * 1000 = 2,5 * 10^ 3

Ich glaube, jetzt beginnt es Ihnen klarzuwerden, dass man auf diese Art wohl alle Zahlen irgendwie darstellen kann. Man dröselt die Zahlen auseinander und bildet ein Produkt, von dem der eine Multiplikator durch 10 teilbar ist (durch die Basis unseres normalen Zahlensystems).

Genauer gesagt: Ein Faktor (also in den Beispielen 1000 oder 100) ist darstellbar als Potenz von 10. Der andere Faktor (in den Beispielen 1, 2 oder 2,5) wird Mantisse (vom lateinischen manitissa = Zugabe, Anhang, Schleppe) genannt. Sehen wir uns noch mal 2500 an:

2500 = 2,5 * 1000 = 2,5 * 10 ^ 3
= 25 * 100 = 25 * 10 ^ 2
= 250 * 10 = 250 * 10 ^ 1
= 2500 * 1 = 2500 * 10 ^ 0

Das Letzte war nur der Vollständigkeit halber, denn irgendeine Zahl hoch 0 ist immer 1. Man kann aus der 2500 auch Folgendes machen:

2500 = 0,25 * 10000 = 0,25 * 10 ^ 4
oder = 0,025 * 100000 = 0,025 * 10 ^ 5

und so weiter. Oder anders herum:

2500 = 25000 * 0,1 = 25000 * 10 ^ -1
= 250000 * 0,01 = 250000 * 10 ^ -2

und so weiter. Dabei bedeutet:

10 ^ -2 = 1 / 10 ^ 2 = 0,01

Man kann sich das merken, indem man die Anzahl der Stellen zählt, um die man das Komma verschiebt. Diese Anzahl addiert man dann zur Hochzahl. Zur Erläuterung:

0,12345 = 1,2345 * 10 ^ -1

Wir haben das Komma um eine Stelle nach rechts gerückt, weshalb wir die Hochzahl -1 schreiben müssen (vorher war da nämlich unsichtbar die Hochzahl 0, und 10 ^ 0 = 1).

0,12345 = 123,45 * 10 ^ -3

Hier wurde das Komma um drei Stellen nach rechts verschoben. Daher der Exponent -3. Sie sehen folgenden Zusammenhang:

Komma eine Stelle nach rechts verschoben: Exponent + (-1). Zum Beispiel:
0,1234 * 10 ^ -2 = 1,234 * 10 ^ -3

Komma eine Stelle nach links verschoben: Exponent + 1. Zum Beispiel:
3,14 * 10 ^ 12 = 0,314 * 10 ^ 3

Verstehen Sie nun, warum man diese Art der Zahlendarstellung Fließkomma- oder Gleitkommazahlen nennt?

Vielleicht sehen Sie aber noch nicht den Sinn der Fließkommazahlen ein. Dazu gebe ich Ihnen zwei einsichtige Beispiele. Der Atomkern eines Heliumatoms wiegt etwa (halten Sie sich fest): 0,000 000 000 000 000 000 000 000 006 643 kg.

Sehr unbequem, diese ganzen Nullen immer mitzuschleppen. Wir verschieben deshalb das Komma um 27 Stellen nach rechts und schreiben dann 6,643 * 10 ^ -27 kg.

2. Beispiel: Wir haben einen Ballon mit diesem Gas gefüllt. Bei normalen Temperatur- und Luftdruckbedingungen befinden sich in einem Kubikzentimeter im Ballon ungefähr (noch mal festhalten!):

26 900 000 000 000 000 000 Heliumatome. Wieder eine recht unangenehme Null-Schlepperei.

Wir verschieben das Komma um 19 Stellen nach links und erhalten 2,69 * 10 ^ 19 Heliumatome. Fein, nicht wahr? Abgesehen von der höheren Bequemlichkeit: Der Com-

puter müsste allerhand Speicherplatz zur Handhabung der vielen Nullen bereitstellen. Mit BCD-Zahlen könnten wir zwar jede Zahl erfassen, hätten aber immer unterschiedlich viele Bytes zu verarbeiten.

Wenn wir Fließkommazahlen verwenden, können wir - wie Sie noch sehen werden - jede (na, sagen wir mal: fast jede) Zahl in der gleichen Anzahl Bytes aufbewahren.

Vom BASIC her kennen Sie Fließkommazahlen auch (hier wird das Komma jedoch, entsprechend der amerikanischen Schreibweise, durch den Punkt ersetzt). Das sind die, wo man zum Beispiel schreibt 6.02E23 oder 6.02E + 23, was dann bedeutet: 6,02 * 10 ^ 23.

»E« steht dort für Zehnerexponent. Durch die Art, wie Fließkommazahlen im normalen Computerdasein gespeichert werden, ergeben sich obere und untere Grenzen. Die höchste in BASIC verarbeitbare Zahl ist im C 64 + 1.70141183 * 10E38.

Größere Zahlen verursachen in BASIC einen »OVERFLOW ERROR«. Was in Maschinensprache mit größeren Zahlen geschieht, ist weitgehend unsere Sache. Die dem Betrag nach kleinste verarbeitbare Zahl ist ± 2.93873588 * 10E-39.

In BASIC arbeitet bei Unterschreitung der Computer einfach mit einer Null weiter. Für die Behandlung in Maschinensprache sind ebenfalls wir als Programmierer verantwortlich.

Für diesmal sei's genug der Zahlenspiele; in der nächsten Ausgabe werden wir uns weiter mit Fließkommazahlen befassen.

Die USR-Funktion

Wieder einmal soll uns das Zusammenspiel von BASIC und Maschinensprache beschäftigen. Einen Aufruf von Maschinenroutinen - nämlich den mit SYS - haben wir schon kennen gelernt.

Wir POKEten die zu übergebenden Werte an die Abrufspeicherstellen. Bei diesen Werten hat es sich um einfache Integerzahlen gehandelt, zum Beispiel die Anzahl der Glieder einer zu summierenden arithmetischen Reihe. Was tun wir aber, wenn wir Fließkommavariablen an ein Maschinenprogramm übermitteln wollen?

Gewiss, werden Sie sagen, lernen wir das ja in den nächsten Folgen und können dann entsprechende POKE-Kommandos geben. Damit haben Sie auch recht, nur ist das dann der »harte« Weg. Es gibt auch einen problemlosen »weichen« Weg, nämlich das USR-Kommando.

USR ist ein BASIC-Befehl und rührt her von »User callable machine language SubRoutine«, also »durch den Benutzer aufrufbares Maschinensprachunterprogramm«. Darin liegt eigentlich noch nichts Neues gegenüber dem SYS-Befehl.

Im Gegensatz zu SYS - wo das Argument die Einsprungadresse des Maschinenprogramms ist - übergibt USR als Argument eine beliebige Fließkommavariable in festgelegter Form an eine sehr nützliche Speicherstellenkombination, den Fließkomma-Akkumulator 1, von uns künftig einfach FAC genannt.

Der FAC belegt die Speicherstellen 97 bis 102 ($61 bis $66). Wenn das eventuell in BASIC benötigte Ergebnis dort auch in der vorgeschriebenen Form abgelegt wird, kann es im BASIC-Programm weiterverwendet werden.

Keine Angst, dazu kommen wir bei der weiteren Behandlung der Fließkommazahlen noch ganz ausführlich zu sprechen. Heute soll uns das noch nicht belasten. Als Argument kann man nämlich auch irgendeine bedeutungslose Größe, einen so genannten »Dummy« angeben, der dann gar nicht weiterverwendet wird. Der USR-Befehl dient in diesem Fall lediglich dem bequemen Ansteuern eines Maschinenprogramms.

Woher weiß unser Computer beim USR-Befehl, welche Maschinenroutine er im 64-KByte-Speicher bearbeiten soll? Beim SYS-Befehl ist das klar - das Argument sagt es: SYS 24345 lässt den Programmzähler auf dez. 24345 zeigen. Aber wenn wir einge-

ben: USR (24345), dann packt der Computer die Zahl 24345 als Fließkommavaria-ble in den FAC und meldet dann einen »SYNTAX ERROR«.

Das liegt daran, dass der BASIC-Interpreter beim USR-Befehl einen der oben kennen gelernten indirekten Sprünge ausführt: JMP (311).

$311 / 312 (dezimal 785 / 786) ist also ein Vektor, und der weist im Normalfall zu einer Routine, die den »SYNTAX ERROR« ausgibt (dez. 45 640). Bevor wir also den USR-Befehl geben, müssen wir in diesen Vektor die Startadresse unserer Maschinenroutine schreiben:

```
dez. 24345 = $5F19
   LSB        $19 = dez. 25 in
                    Speicher 785
                    mit
                    POKE 785,25
   MSB        $5F = dez. 95 in
                    Speicher 786
                    mit
                    POKE 786,95
```

Jetzt weiß der Computer, wohin er beim USR-Aufruf springen soll, und solange, bis wir den Vektor wieder ändern, führt er bei jedem USR-Befehl unser bei 24345 stehendes Maschinenprogramm aus. Wir müssen nur noch dafür sorgen, dass dort dann auch wirklich eines anfängt. Ein Beispiel werden wir nachher noch behandeln.

Der harte Kern: noch mal Speicherfragen

Die Struktur des C 64-Speichers ist vereinfacht schon zu Beginn dieses Kurses gezeigt worden (Seite 16).

Dabei tauchten zwei ROM-Bereiche auf, die wir BASIC-Interpreter und Betriebssystem genannt haben. Diese Unterteilung ist nicht ganz korrekt. Wenn Sie über ein ROM-Listing verfügen und beispielsweise das Ende des ROM-Bereichs von $A000 bis $BFFF sowie den Anfang des oberen ROM ($E000 bis $FFFF) untersuchen, dann stellen Sie fest, dass ab dez. 49087 ($BFBF) die BASIC-Funktion »EXP« bearbeitet wird. Der letzte Befehl vor $C000 beendet diese Funktion aber nicht etwa, sondern dort steht: JMP E000.

Tatsächlich läuft ab $E000 bis $E042 die Bearbeitung der EXP-Funktion munter weiter, und auch danach finden sich allerlei BASIC-Befehle (SIN, COS und so weiter).

Da liegt also keine klare Trennung vor, sondern ein Mischmasch. Wir sollten uns vielleicht angewöhnen, statt vom Interpreter und dem Betriebssystem vom unteren und oberen ROM-Bereich zu sprechen.

Eine andere Unterscheidung ist dagegen sinnvoll: Wie einige Besitzer neuerer Commodore 64 sicherlich bemerkt haben, sind Teile der ROM-Routinen im Laufe der Zeit verändert worden. Hauptsächlich geht es bei den aktuellen Neuerungen dieser internen Maschinenprogramme um die Farbgebung der Zeichen.

Man kann eigentlich nie so recht wissen, was den Software-Planern von Commodore noch alles einfällt. Jedenfalls können deren Ideen manchmal recht dramatische Folgen haben, nämlich dann, wenn Sie ein fabelhaftes Maschinenprogramm gebaut haben, welches ROM-Routinen direkt verwendet. Der Programmierer spielt auf diese Weise eine müde Form des russischen Roulettes.

Glücklicherweise halten sich die Änderungen in Grenzen, und wir dokumentieren unsere Programme ja auch immer gut (Sie etwa nicht?). Notwendige Umbauten können also leicht vonstatten gehen.

Ganz ohne ROM-Routinen-Verwendung kommt man eigentlich kaum aus. Es gibt aber einen ROM-Bereich, für den Commodore verspricht, keinerlei Änderungen durchzuführen: die KERNAL-Sprungtabelle.

Das ist ein Programmbereich ($FF81 bis $FFF5), in dem 39 JMP-Befehle enthalten sind (zum Teil in absoluter, aber auch in indirekter Adressierung). Jeder dieser Sprungbefehle weist auf die Einsprungadresse eines Maschinenprogramms. Da finden sich alle wichtigen Ein- / Ausgabe-Operationen,

Tab. 13:
Kernal-Routinen

Adresse			
HEX	DEZ	Name	Funktion
FF81	65409	CINT	Prüfen der TV-Norm, Berechnung der Taktfrequenz
FF84	65412	IOINIT	Ein- / Ausgabe-Reset
FF87	65415	RAMTAS	Prüfen auf freies BASIC-RAM
FF8A	65418	RESTOR	Initialisieren der I / O-Vektoren
FF8D	65421	VECTOR	Lesen und Setzen der I / O-Vektoren
FF90	65424	SETMSG	Setzen des Ausgabe-Modus'
FF93	65427	SECOND	Ausgeben der Sekundär-Adresse nach LISTEN
FF96	65430	TKSA	Ausgabe der Sekundär-Adresse nach TALK
FF99	65433	MEMTOP	Lesen / Setzen des Speicherendes
FF9C	65436	MEMBOT	Lesen / Setzen des Speicheranfangs
FF9F	65439	SCNKEY	Abfragen der Tastatur
FFA2	65442	SETTMO	Setzen der Time-Out-Flagge
FFA5	65445	ACPTR	Zeichen vom seriellen Port in Akku lesen
FFA8	65448	CIOUT	Zeichen vom Akku auf seriellen Port ausgeben
FFAB	65451	UNTLK	Sendet UNTALK an seriellen Bus
FFAE	65454	UNLSN	Sendet UNLISTEN an seriellen Bus
FFB1	65457	LISTEN	Sendet per seriellen Bus LISTEN an Geräte
FFB4	65460	TALK	Sendet per seriellen Bus TALK an Geräte
FFB7	65463	READST	Liest I / O-Status in den Akku
FFBA	65466	SETLFS	Festlegung der Parameter für OPEN
FFBD	65469	SETNAM	Festlegung des File-Namens
FFC0	65472	OPEN	Öffnet spezifiziertes File
FFC3	65475	CLOSE	Schließt spezifiziertes File
FFC6	65478	CHKIN	Öffnet einen Eingabekanal
FFC9	65481	CHKOUT	Öffnet einen Ausgabekanal
FFCC	65484	CLRCHN	Schließt Ein- und Ausgabekanäle
FFCF	65487	CHRIN	Holt ein Zeichen vom aktiven Eingabekanal in den Akku
FFD2	65490	CHROUT	Sendet Akku-Inhalt auf aktiven Ausgabekanal
FFD5	65493	LOAD	LOAD und VERIFY von Programmen
FFD8	65496	SAVE	Speichern von Programmen
FFDB	65499	SETTIM	Uhrzeit setzen
FFDE	65502	RDTIM	Uhrzeit lesen
FFE1	65505	STOP	STOP-Taste abfragen
FFE4	65508	GETIN	Zeichen aus dem Tastaturpuffer in den Akku lesen
FFE7	65511	CLALL	Schließen aller Kanäle und Files
FFEA	65514	UDTIM	Uhr um 1/60 Sekunde weiterzählen
FFED	65517	SCREEN	Lesen des Bildschirmformats
FFF0	65520	PLOT	Lesen / Setzen der Cursor-Position
FFF3	65523	IOBASE	Lesen der Startadresse der Ein- und Ausgabebausteine

Systemtakt- und Uhrsteuerungen und Anderes mehr. Wir werden uns nach und nach damit vertraut machen.

In Tabelle 13 sind die Kernal-Adressen und ihre Funktion aufgeführt. Manche davon können ohne jede Vorbereitung benutzt werden, andere brauchen bestimmte Routinen oder Angaben, um sinnvoll zu arbeiten.

Die Absicht von Commodore ist es, dass jeder Aufruf von zum Beispiel FFD2 die Ausgabe eines Zeichens bewirkt, und zwar unabhängig davon, welchen Computer in welcher Version wir benutzen. Das Programm, welches diese Zeichenausgabe letztendlich ausführt, kann sich ändern und in ganz andere Speicherbereiche gelegt werden. An der Stelle $FFD2 wird aber immer ein JMP mit der Einsprungadresse stehen. Leider ist diese Sprungtabelle viel zu knapp gehalten. Es gibt so viele interessante ROM-Routinen, die wir alle ohne diese schöne Sicherheit anspringen müssen.

Die Urzelle eines Programmprojekts

Wir sind jetzt so weit, dass wir die Urzelle eines Programmprojekts, welches uns eine lange Zeit begleiten wird, aufbauen können.

Wir wollen etwas unter den Teppich kehren. Der Teppich, das sind die uns bislang nicht zugänglichen RAM-Bereiche unter den ROMs. Haben Sie das nicht auch schon mal erlebt, dass Sie während einer Programmarbeit plötzlich feststellen, dass Sie zum Beispiel für eine Zwischenrechnung ein weiteres Programm benötigen, oder Sie wälzen Listen und denken sich, ein kleiner Hilfsbildschirm wäre jetzt von Nutzen, oder ...

Mit diesem heute zu startenden Programm wäre all das und noch viel mehr realisierbar. Es soll auf einfache Weise beliebige Speicherbereiche unters ROM schieben und sie wieder hervorholen können.

Natürlich braucht die Entwicklung dieses Projekts einige Zeit, zumal wir noch Vieles lernen müssen. Deswegen sind wir in dieser ersten Urzelle noch sehr eingeschränkt: Wir verschieben zuerst einmal nur eine Bildschirm-Kopfzeile unter den oberen ROM-Bereich. Auch in dieser einfachsten Version gibt es noch einige Programmteile, die Sie erst nach der nächsten Ausgabe verstehen werden. Aber irgendwann müssen wir ja mal anfangen, Nagel mit Köpfen zu machen.

Unser Maschinenprogramm soll durch die USR-Funktion aufgerufen werden. Wie wir es in dieser Ausgabe gelernt haben, muss deshalb vor dem ersten Aufruf eine Initialisierung durch Belegen des USR-Vektors mit unserer Startadresse stattfinden.

Die Startadresse soll $02B6 (dez. 694) sein, denn dort gibt es einen freien RAM-Bereich bis inklusive $02FF (dez. 767), der weder andere Programme noch Kassettenoperationen stört. Das MSB $02 ist dezimal auch 2 und wird nach 786 gePOKEt: POKE 786,2.

Das LSB $B6 ist dezimal 182 und soll in 785 geschrieben werden: POKE 785,182.

Damit ist der USR-Vektor gestellt, und wir brauchen uns nicht mehr weiter darum zu kümmern: Jeder USR-Aufruf wird nun den Start des Programms bewirken. Nun zum Programm selbst. In Abbildung 12 finden Sie ein Flussdiagramm dazu.

Zunächst konstruieren wir den Teil, der die erste Bildschirmzeile nach $E000 und folgende Speicherstellen schiebt. Das X-Register verwenden wir als Index und laden es mit dez. 40 = $27.

Schalten Sie also den SMON ein und starten Sie den Assembler mit A 02B6. Dann geben Sie ein:

```
02B6 LDX #27
```

Nun packen wir das letzte Zeichen der obersten Bildschirmzeile in den Akku:

```
02B8 LDA 0400,X
```

In das Y-Register legen wir die dazugehörige Farbe aus dem Bildschirmfarbspeicher:

```
02BB LDY D800,X
```

Den Akkuinhalt - also die Bildschirminformation - legen wir nach $E000 + $27:

```
02BE STA E000,X
```

Dasselbe tun wir mit dem Farbkode, der ab $E028 + $27 abwärts gespeichert wird. Leider kann man STY nicht X-indiziert-absolut adressieren (siehe Tabelle 11, Seite 58). Deshalb schieben wir zuerst den Y-Registerinhalt in den Akku:

```
02C1 TYA
02C2 STA E028,X
```

Damit ist das letzte Zeichen der Kopfzeile verschoben. Wir zählen das X-Register um 1 herunter:

```
02C5 DEX
```

Der X-Index weist nun auf das vorletzte Zeichen, mit dem sich alles ab $02A9 wiederholt. Wenn das X-Register bis 0 heruntergezählt ist, weist es auf das erste Zeichen der Kopfzeile. Die Schleife muss dann noch einmal durchlaufen werden, und ein weiteres Herabzählen des X-Registers erzeugt $FF, was zum Setzen der N-Flagge führt.

Das ist dann unser Signal, dass die gesamte Kopfzeile übertragen wurde. Die N-Flagge wird durch den BPL-Befehl getestet:

```
02C6 BPL 02B8
```

So weit, so gut. Wir hätten natürlich auch das X-Register von 0 an hochzählen können. Zum Beenden der Schleife wäre dann aber ein CPX-Befehl erforderlich gewesen, der jedes Mal den X-Registerinhalt mit der Zahl $27 vergleicht.

MERKE: Indexregister in Schleifen abwärts zu zählen, kann Rechenzeit einsparen!

Ab $02CE soll der umgekehrte Vorgang, also das Zurückschieben der vorher gespeicherten Kopfzeile in den Bildschirmspeicher geschehen. Das einfachste wäre es sicher-

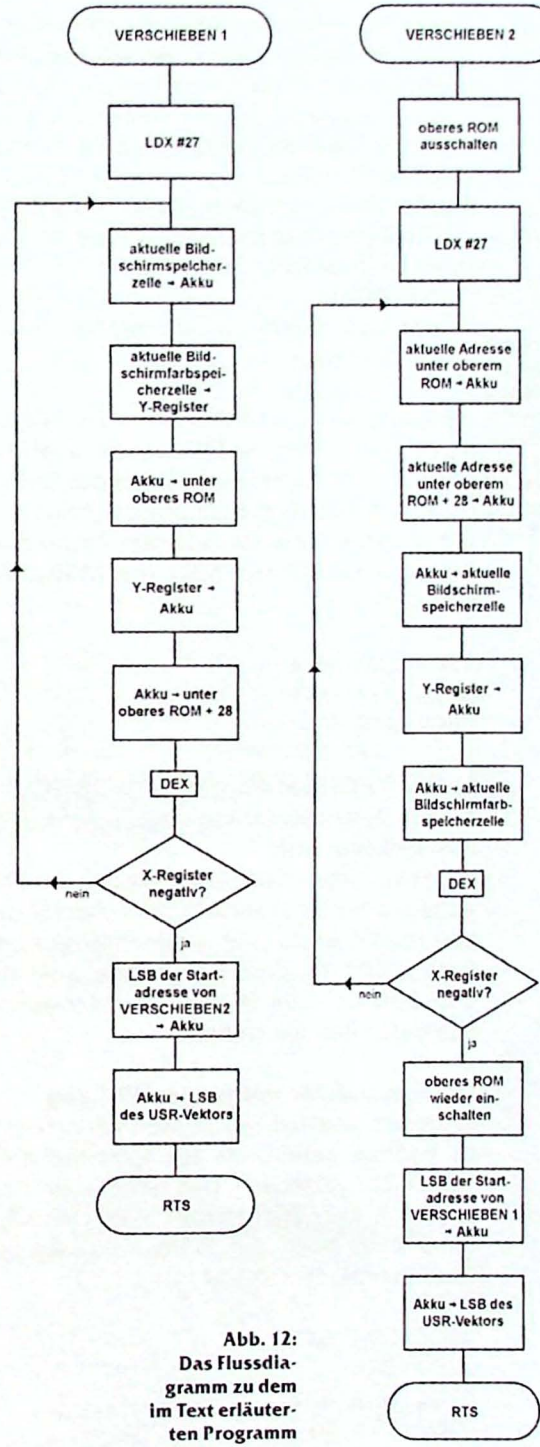


Abb. 12: Das Flussdiagramm zu dem im Text erläuterten Programm

lich, diesen Programmteil mit einem weiteren USR-Kommando zu starten. Das sähe dann so aus:

Erster USR-Befehl schiebt Kopfzeile unter oberes ROM
Zweiter USR-Befehl holt Kopfzeile zurück in Bildschirmspeicher
Dritter USR-Befehl schiebt wieder Kopfzeile unter ROM
Vierter USR-Befehl holt sie wieder zurück und so weiter

Weil aber das Umstellen des USR-Vektors durch POKEs vom BASIC aus lästig ist, tun wir das einfach immer am Ende des betreffenden Maschinenprogrammabschnitts.
Wir schreiben also das LSB der Programmfortführung ($CE) nach $311. Das MSB bleibt unverändert $02.

```
02C8 LDA #CE
02CA STA 0311
02CD RTS
```

Mit dem RTS sind wir wieder im BASIC-Programm gelandet, welches nun normal weiterverarbeitet wird.

Erst ein neues USR-Kommando - im Programm oder im Direktmodus - startet den zweiten Teil unseres Maschinenprogramms (weil in $0311 - dem Einsprungspunkt des USR-Befehls - die Startadresse der auszuführenden Routine steht).

Einfache Befehle mit großer Wirkung

In diesem zweiten Teil müssen wir erst einige Befehle geben, die Sie jetzt vielleicht noch nicht verstehen. Das hängt damit zusammen, dass zum Herauslesen des RAM unter dem ROM das ROM ausgeschaltet werden muss (entspricht POKE 1,53):

```
02CE LDA 01
02D0 PHA
02D1 LDA #35
02D3 STA 01
```

(Der PHA-Befehl dient hier zur Zwischenspeicherung des Akku-Inhalts). Das ist hiermit geschehen, und wir kommen wieder in bekannte Gefilde mit der Ausleseschleife:

```
02D5 LOX #27
02D7 LDY E000,X
02DA LDY E028,X
02DD STA 0400,X
02E0 TYA
02E1 STA D800,X
02E4 DEX
02E5 BPL 02D7
```

Damit ist die gesamte gespeicherte Kopfzeile wieder zurückgeholt, und wir können das ROM wieder einschalten:

```
02E7 PLA
02E8 STA 01
```

Falls nun wieder ein USR-Kommando auftaucht, soll die Kopfzeile wie am Anfang mit dem ersten Programmteil unter das obere ROM gelegt werden. Wir müssen deshalb den USR-Vektor auf $02B6 zurückschreiben:

```
02EA LDA #B6
02EC STA 0311
02EF RTS
```

Das wär's! Wenn nun im Programm oder im Direktmodus wieder ein USR-Befehl auftritt, kann das Ganze von vorne beginnen. In dieser Version wird jedes Mal eine neue Kopfzeile hin und wieder zurück geschoben.

Wenn Sie eine einmal festgelegte Kopfzeile immer wieder benutzen möchten, dann stellen Sie den USR-Vektor einfach nicht mehr zurück: Lassen Sie also die Befehle bei 02EA und 02EC weg. Das Programm endet in dem Fall mit: 02EA RTS

Eine wichtige Bemerkung noch: So bequem der Ort auch ist, an dem unser kurzes Programm steht, er hat einen gravierenden Nachteil: Falls Sie mittels einer RESET-Taste

oder per Software einen BASIC-Kaltstart durchführen, geht unser Programm flöten!

Dieser Speicherbereich wird im Reset-Programm nämlich mit lauter Nullen überschrieben. Deswegen speichern Sie es bitte bald ab.

Damit sind wir für diesmal am Ende. Sie finden noch ein kleines Testprogramm für unsere Verschieberoutine und in Tabelle 14 wie immer eine Zusammenfassung aller wichtigen Daten der neuen Befehle. In der nächsten Folge greifen wir noch einmal das Thema Fließkomma auf, werden die einfachsten und kürzesten Kurzspeicher-Befehle kennen lernen und beginnen mit den leistungsfähigsten Befehlen des 6502, den indirekt-indizierten Befehlen.

```
1 REM **********************************
2 REM *                                *
3 REM * TEST FUER DIE 1. VERSION DES   *
4 REM *       PROGRAMM-PROJEKTS        *
5 REM *        VERSCHIEBEN VON         *
6 REM *       SPEICHERBEREICHEN        *
7 REM *                                *
8 REM * HEIMO PONNATH     HAMBURG 1984 *
9 REM **********************************
10 REM
15 REM +++++ USR-VEKTOR EINSTELLEN +++++
20 REM
25 POKE 785,182 : POKE 786,2
30 REM
35 REM +++++ KOPFZEILE ++++++++++++++++
40 REM
45 PRINT CHR$<147)CHR$<18>"TEST: BILD $0400=1024, FARBE $D800=55296"CHR$(146)
50 PRINT : PRINT : PRINT "DURCH IRGENDEIN USR-KOMMANDO WIRD NUN IM PROGRAMM-MODUS"
55 PRINT "DER ERSTE TEIL DES VERSCHIEBE-PROGRAMMS AUFGERUFEN."
60 PRINT "DIE KOPFZEILE WIRD UNTER DAS OBERE ROM{2 SPACE}KOPIERT."
65 REM
70 REM +++++ 1. USR-AUFRUF ++++++++++++
75 REM
80 A=USR(1)
85 PRINT : PRINT "HIER GESCHIEHT DAS DURCH A=USR(1) IN{4 SPACE}ZEILE 65"
90 PRINT "DABEI IST 1 EIN DUMMY UND MIT A FANGEN{2 SPACE}WIR AUCH NICHTS WEITER AN."
95 PRINT "AUF TASTENDRUCK WIRD DER BILDSCHIRM{2 SPACE}GELOESCHT."
100 REM
105 REM + UEBERSCHREIBEN DER KOPFZEILE +
110 REM
115 POKE 198,0 : WAIT 198,1 : PRINT CHR$(147)
120 REM
125 REM ++++ NEUBEGINN DES PROGRAMMS ++++
130 REM
135 PRINT CHR$(19)"WAS AUCH IMMER JETZT IN DER KOPFZEILE{3SPACE}STEHT, ES WIRD BEIM 2. USR"
140 PRINT "VON DEM ZUVOR DURCH DAS ERSTE USR GE-{3 SPACE}SPEICHERTE UEBERSCHRIEBEN"
145 PRINT : PRINT "WENN SIE JETZT EINE TASTE DRUECKEN ..."
150 POKE 198,0 : WAIT 198,1
155 REM
160 REM +++++ 2. USR-AUFRUF ++++++++++++
165 REM
170 A=USR(1) : PRINT
175 PRINT "IST DIE ALTE KOPFZEILE ZURUECK IN DEN{3 SPACE}BILDSCHIRMSPEICHER GESCHOBEN."
180 END
```

Progr. 1: Test und Demonstration der Verschieberoutine. Das Programm zeigt das Ein- und Ausschalten einer Kopfzeile auf dem Bildschirm.

Tab. 14: Zusammenfassung aller wichtigen Daten der neuen Befehle

Befehlswort	Adressierung	Byte-Anzahl	Kode		Dauer in Taktzyklen	Beeinflussung von Flaggen
			HEX	DEZ		
LDA	absolut, X	3	BD	189	4	N, Z
	0-page-absolut,X	2	B5	181	4	N, Z
	absolut, Y	3	B9	185	4	N, Z
LDX	absolut, Y	3	BE	190	4	N, Z
	0-page-absolut,Y	2	B6	182	4	N, Z
LDY	absolut, X	3	BC	188	4	N, Z
	0-page-absolut,X	2	B4	180	4	N, Z
STA	absolut, X	3	9D	157	5	/
	absolut, Y	3	99	153	5	/
	0-page-absolut,X	2	95	149	4	/
STX	0-page-absolut,Y	2	96	150	4	/
STY	0-page-absolut,X	2	94	148	4	/
INC	absolut, X	3	FE	254	7	N, Z
	0-page-absolut,X	2	F6	246	6	N, Z
DEC	absolut, X	3	DE	222	7	N, Z
	0-page-absolut,X	2	D6	214	6	N, Z
ADC	absolut, X	3	7D	125	4	N, V, Z, C
	absolut, Y	3	79	121	4	N, V, Z, C
	0-page-absolut,X	2	75	117	4	N, V, Z, C
SBC	absolut, X	3	FD	253	4	N, V, Z, C
	absolut, Y	3	F9	249	4	N, V, Z, C
	0-page-absolut,X	2	F5	245	4	N, V, Z, C
CMP	absolut, X	3	DD	221	4	N, Z, C
	absolut, Y	3	D9	217	4	N, Z, C
	0-page-absolut,X	2	D5	213	4	N, Z, C
BIT	absolut	3	2C	44	4	N, V, Z
	0-page-absolut	2	24	36	3	N, V, Z
CLV	implizit	1	B8	184	2	V
NOP	implizit	1	EA	234	2	/
TAX	implizit	1	AA	170	2	N, Z
TAY	implizit	1	A8	168	2	N, Z
TXA	implizit	1	8A	138	2	N, Z
TYA	implizit	1	98	152	2	N, Z
JMP	absolut	3	4C	76	3	/
	indirekt	3	6C	108	5	/
JSR	absolut	3	20	32	6	/

Assembler ist keine Alchemie
Folge 7

Erschienen in: 64'er 3/1985
Verantwortlicher Redakteur: Georg Klinge
Autor: Heimo Ponnath

In diesem Teil werden wir drei Themen besprechen: den Stapel und die indirekte Adressierung. Ferner zeigen wir an einem ausführlichen Beispiel, wie Text auf dem Bildschirm und auf dem Drucker ausgegeben werden kann.

Die Assembler-Folge in der letzten Ausgabe hat einige Fragen offen gelassen, die dieses Mal beantwortet werden sollen: Die Sache mit dem Stapel wird geklärt, und Sie werden Befehle zu gebrauchen lernen, die uns Stapeloperationen ermöglichen.

In unserem ersten Programmprojekt-Teil gab es für Sie unverständliche Sequenzen, die mit der Speicherstelle 1 zu tun hatten. Auch das werden Sie diesmal verstehen.

Sie beherrschen inzwischen fast alle Arten der Adressierung - nach dieser Folge fehlt keine mehr. Ich habe Ihnen hoffentlich das Wasser im Munde zusammenlaufen lassen mit der Liste aller Kernal-Routinen: Nun sollen uns die ersten davon munden.

Wir stapeln

In der vorangegangenen Ausgabe haben wir beim JSR-Befehl schon den Stapel etwas kennen gelernt. Aber so ganz genau wissen wir ja noch nicht, was das ist. Deswegen jetzt mal im Detail:

Der Stapel, auch Prozessorstack genannt, ist der Speicherbereich von dezimal 256 ($100) bis dezimal 511 ($1FF), der direkt von unserer CPU verwaltet wird. Das ist also die gesamte Page 1. Ähnlich wie bei der String-Verwaltung geschieht auch hier das Füllen von oben nach unten. Das erste Byte, welches in den Stack geschoben wird, kommt also nach $1FF, das nächste nach $1FE und so weiter. Voll ist der Stapel, wenn auch $100 besetzt wurde (siehe Abbildung 13).

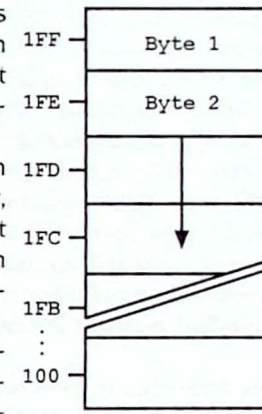


Abb. 13: So wird der Stapel gefüllt.

Warum heißt das Ding nun eigentlich Stapel? Das erklärt sich aus dem Zugriffs-Prinzip.

Man spricht von einer LIFO-Struktur, von »Last In - First Out«, zu deutsch »zuletzt hinein - zuerst heraus«.

Das zuerst hineingebrachte Byte befindet sich am Speicherboden ($1FF), das zuletzt eingebrachte an der Speicherspitze. Stellen Sie sich einen Stapel Akten vor (Abbildung 14).

Offensichtlich wurde der vierte Aktenordner zuletzt auf den Stapel gesteckt. Er kann zuerst heruntergeholt werden. An die Akte 1 kommen wir erst heran, wenn alle anderen heruntergenommen worden sind.

Genauso verhält es sich mit dem Prozessorstack: Um an das unterste Byte des Stapels heranzukommen, müssen erst Byte für Byte die darüberliegenden (nach Abbildung 13 eigentlich die darunterliegenden) weggeschafft werden.

Mit dem Prinzip des Stapelspeichers werden Sie sich auskennen, wenn Sie schon mal andere Programmiersprachen als BASIC aus-

probiert haben: In Forth beispielweise operieren Sie ständig mit Stapeln.

Der Stapel: das Gedächtnis des Prozessors

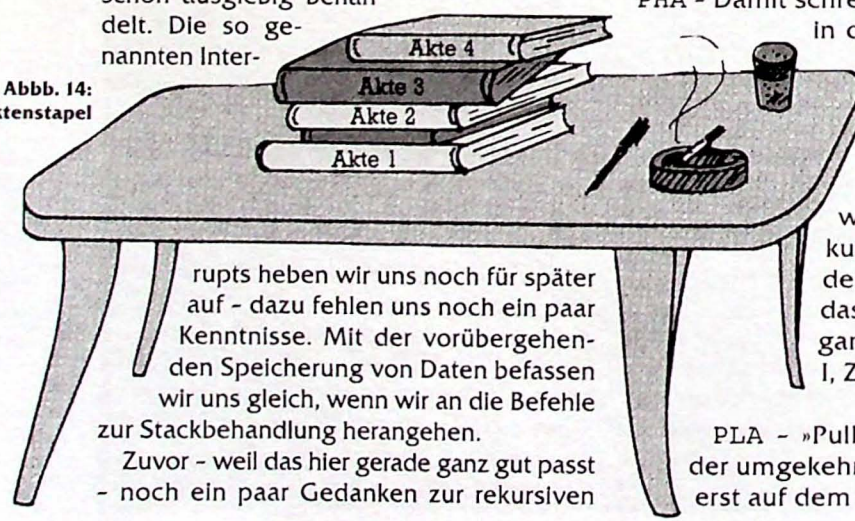
Damit wir - und der Prozessor - den Überblick über den Stack behalten, gibt es dankenswerterweise noch einen Stapelzeiger (stackpointer), der jeweils auf den nächsten freien Platz des Stapels weist. Da gibt's nun aber ein kleines Problem: Der Stapel belegt die komplette Seite 1.

Ein Stapelzeiger, der auf zum Beispiel $01FE zeigen soll, müsste das MSB (also 01) und das LSB (also FE) in zwei Bytes lagern. Der Stapelzeiger ist aber nur 8 Bit groß ...

Freundlicherweise sorgt unser Mikroprozessor automatisch für das neunte Bit. Der Zeiger zählt also immer von $FF an rückwärts bis $00 und weist dabei von $1FF bis $100. Der Stack hat in unserem Computer drei Aufgaben zu erfüllen:

1. Organisation von Unterprogramm-Adressen
2. Zwischenspeicherung bei Unterbrechungen (Interrupts)
3. vorübergehende Datenspeicherung

Die Rolle des Stapels bei Unterprogramm-Aufrufen haben wir in der letzten Folge schon ausgiebig behandelt. Die so genannten Interrupts heben wir uns noch für später auf - dazu fehlen uns noch ein paar Kenntnisse. Mit der vorübergehenden Speicherung von Daten befassen wir uns gleich, wenn wir an die Befehle zur Stackbehandlung herangehen.



Abbb. 14: Der Aktenstapel

Zuvor - weil das hier gerade ganz gut passt - noch ein paar Gedanken zur rekursiven Programmierung. Gemeint ist damit eine Programmstruktur, in der sich ein Unterprogramm selbst aufruft. Auch GOSUB-Befehle in BASIC bewirken Einträge der Rücksprungadressen im Stapel. Auf diese Weise ergibt sich für unseren Computer eine begrenzte Verschachtelungstiefe bei Unterprogrammaufrufen. Diese wird bei Rekursion besonders schnell erreicht, und das bewirkt die Ausgabe einer »OUT OF MEMORY«-Fehlermeldung.

Aktives Stapeln mit PHA, PLA, PHP, PLP, TSX und TXS

Mit dem Stapel haben wir 256 Speicherplätze für eine schnelle Zwischenspeicherung aller möglichen Daten zur Verfügung. Weil der 6510 (und natürlich auch der 6502) diesen Speicherbereich wie die Zeropage behandelt, geht das Speichern sehr schnell. Man muss nur immer die spezielle LIFO-Struktur berücksichtigen.

Im Grunde braucht man eigentlich nur zwei Befehle: Etwas auf den Stapel schieben (in der Literatur oft als Push-Befehl bezeichnet) und etwas herunterziehen - das nennt man dann Pull- oder auch Pop-Befehl.

Unser Prozessor kennt insgesamt sechs auf den Stapel wirkende Anweisungen:

PHA - Damit schreibt man den Akku-Inhalt in den Stapel (»PusH Accumulator«).

Der Stapelzeiger wird automatisch eine Position heruntergezählt (er rechnet ja von $FF an abwärts!). Der Inhalt des Akkus wird dabei nicht verändert. Deswegen bleibt auch das Status-Register (also die ganzen Flaggen: N, V, B, D, I, Z und C) unbeeinflusst.

PLA - »PulL Accumulator«. Das ist der umgekehrte Weg: Das, was zuoberst auf dem Stapel liegt, wird in den

Akku geschrieben. Dadurch wird ein Stapelplatz frei, was den Stapelzeiger veranlasst, um 1 zu wachsen. Weil das, was da in den Akku geladen wird, 0 sein kann oder auch negativ (also mit gesetztem Bit 7), wird unter Umständen auch die N- oder die Z-Flagge verändert.

Weniger mit Datenzwischenspeicherung haben die anderen Befehle zur Stapel-Manipulation zu tun:

PHP - Das steht für »PusH Processor status«, also »schiebe das Prozessor-Status-Register auf den Stapel«. Der aktuelle Flaggenstand kann damit aufbewahrt werden. Das Status-Byte ändert seinen Inhalt dabei ebensowenig wie der Akku bei PHA. Auch hier wird der Stapelzeiger freundlicherweise um 1 herabgezählt.

PLP - »PuLl Processor status«, »hole den Prozessor-Status vom Stapel«, ist der umgekehrte Befehl, der (wie bei PLA in den Akku) den Wert, der zuoberst im Stapel liegt, in das Flaggen-Register schreibt. Da sollte man höllisch aufpassen, was man damit einlädt: Das ist eine feine Gelegenheit für den Computer, abzustürzen. Der Stapelzeiger wird - wie gehabt - um 1 erhöht.

Nicht direkt mit dem Stapel, sondern mit dem Stapelzeiger befassen sich die beiden folgenden Befehle:

TSX - »Transfer Stackpointer into X«, zu deutsch »schiebe den Stapelzeiger ins X-Register«, eröffnet die Möglichkeit, den Stapelzeiger zu lesen. Dabei bleibt er selbst unverändert erhalten. Weil nun im X-Register alle Werte zwischen $FF und 0 auftreten können, werden auch die Flaggen beeinflusst (N- und Z-Flagge).

TXS - Den umgekehrten Weg geht »Transfer X into Stackpointer« = »übertrage X-Register-Inhalt in den Stapelzeiger«. Das ist der einzige Befehl, der es erlaubt, den Stapelzeiger mit einem von uns kontrollierten Wert zu laden. Der Inhalt des X-Registers bleibt dabei unverändert, demzufolge interessieren sich auch die Flaggen nicht dafür.

Alle sechs Anweisungen bestehen nur aus einem Byte und sind implizit adressiert. Die Stapelzeiger-Befehle TXS und TSX benötigen zwei Taktzyklen, die Push-Befehle je drei und die Pull-Befehle vier Taktzyklen zur Bearbeitung.

Es ist etwas schwierig, Stapel-Operationen direkt zu verfolgen. Die meisten Assembler - so anscheinend auch der SMON - gebrauchen ebenfalls diesen Speicherbereich.

Verlangt man beispielsweise mit dem SMON-Kommando M 0100 01FF eine Darstellung des Stapelinhalts, dann findet man eine ganze Menge Spuren der Arbeit des Assemblers. Versucht man die zu löschen oder zu überschreiben, zum Beispiel mit dem nachfolgenden kleinen Programm, dann hat der Assembler die Mühe schon wieder zu Nichte gemacht, wie man durch erneutes M 0100 01FF schnell sehen kann.

Dieses kleine Programm soll unterhalb des durch den Stapelzeiger bezeichneten Bereichs 32 Nullen in den Stapel schreiben:

```
8000 LDA #00
8002 TSX        Der Stapelzeiger wird
                ins X-Register gerettet.
8003 LDY #20
8006 PHA        Wir schieben eine Null
                auf den Stapel.
8006 DEY
8007 BNE 8005
8009 TXS        Nach 32 Eintragungen
                von Nullen stellen wir
                den alten Stapelzeiger
                wieder her.
800A BRK
```

Erneutes Kommando M 0100 01FF zeigt keine Nullen. Erst wenn wir an Stelle des TXS in Zeile 8009 ein BRK schreiben, den Stapelzeiger also nicht zurückschreiben, erscheinen unsere Nullen.

Sieht man genau hin, dann stellt man fest, dass unterhalb des durch den Stapelzeiger bezeichneten Bereichs genau der gleiche Inhalt zu finden ist wie vorher, nur eben mit dem Stapelzeiger verschoben.

Ganz konnte ich dieses Rätsel noch nicht lösen, muss ich gestehen, aber für den Gebrauch des Stapels ändert sich dadurch für uns nichts.

Worauf muss man bei Stapeloperationen achten? Ganz einfach: Zwischen dem Ablagern eines Werts auf dem Stapel und dem Zurückholen muss für jeden Push-Befehl ein Pull-Befehl vorhanden sein, für jedes weitere PHA ein PLA, für jedes JSR ein RTS.

Nur wenn wir auf diese Symmetrie der Push- und der Pull-Befehle achten (und wie Sie noch aus der vorherigen Ausgabe wissen, sind ja JSR und RTS ebenfalls dazuzurechnen), können wir sicher sein, dass der Stapelzeiger zum Zeitpunkt des Rückholens eines Werts vom Stapel auch wirklich darauf deutet. Wenn man also nicht ganz genau weiß, wie der verwendete Assembler den Stapel nutzt, sollte man auf Operationen mit den Befehlen TSX und TXS verzichten.

Nun können Sie schon einen Teil der bislang unbekannten Programmsequenz aus der letzten Folge verstehen. Im zweiten Programmteil hatten wir mit

```
02CE LDA 01
02D0 PHA
```

den Inhalt der Speicherstelle 01 in den Akku geladen und auf den Stapel geschoben. Später - nach einigen weiteren Operationen - wurde dann dieser Speicherinhalt wiederhergestellt durch

```
02E7 PLA
02E8 STA 01
```

Was aber hat es mit dieser Speicherstelle 01 auf sich? Das soll nun als Nächstes erklärt werden.

Tab. 15: Die Tabelle zeigt, welche Bausteine bei verschiedener Belegung der Bits 0 bis 2 des Prozessorports (Speicherstelle 1) eingeschaltet sind.

Speicherstelle 1			$A000 - $BFFF	$D000 - $DFFF	$E000 - $FFFF
Bits 2	1	0			
1	1	1	BASIC	I/O	Kernal
1	1	0	RAM	I/O	Kernal
1	0	1	RAM	I/O	RAM
1	0	0	RAM	RAM	RAM
0	1	1	BASIC	Zeichen	Kernal
0	1	0	RAM	Zeichen	Kernal
0	0	1	RAM	Zeichen	RAM
0	0	0	RAM	RAM	RAM

Sein oder Nichtsein:
Das Rätsel des Prozessorports

Der Commodore 64 hat 64 KByte an RAM zu bieten. Außerdem aber verfügen wir beim normalen Programmieren über weitere 24 KByte, in denen das Betriebssystem, der BASIC-Interpreter, Ein- und Ausgabebausteine und der Zeichenspeicher stecken. Wie Sie aus der ersten Assemblerfolge wissen, umfasst der Adressbus aber nur 16 Bits, was uns lediglich 65 536 Speicherzellen, also 64 KByte adressieren lässt.

Des Rätsels Lösung liegt darin, dass einige Adressenbereiche mehrfach belegt sind. Man kann das vergleichen mit dem Trick des Kastens mit dem doppelten Boden. Welcher Kasteninhalt gerade dem Prozessorzugriff offen steht, wird durch den Prozessorport, das sind die Speicherstellen 00 und 01, gesteuert.

Dr. Helmuth Hauck hat in seiner Serie »Memory Map mit Wandervorschlägen« (64'er 11 / 1984, Seite 135 ff.) die genaue Funktion jedes Bits dieser beiden Speicherstellen erklärt. Wer noch mehr wissen möchte - auch über die Wirkungsweise der beiden Leitungen »Game« und »Exrom« - sollte das nachlesen im »Commodore 64 Programmer's Re-

ference Guide« ab Seite 260. Für uns als angehende Assembler-Alchemisten ist die Speicherstelle 1 aber so wichtig, dass wir hier noch mal ganz kurz darauf eingehen.

Die Speichersteuerfunktionen haben die Bits 0 bis 2 der Speicherstelle 1. Je nach Belegung dieser Bits gestaltet sich die 64-KByte-Landschaft unseres Computers wie in Tabelle 15 gezeigt.

Was können wir als Maschinen-Programmierer mit dieser Kenntnis anfangen? Theoretisch stehen uns für unsere Programme damit 64 KByte offen. Praktisch werden wir nur in den seltensten Fällen auf die Ein- und Ausgabe-Bausteine verzichten können.

Lassen wir ein reines Maschinenprogramm laufen, ohne jeglichen Rückgriff auf Interpreter oder Betriebssystem, dann haben wir immerhin noch zirka 60 KByte zur freien Verfügung. Benutzen wir Routinen aus diesen beiden ROM-Bausteinen, dann müssen wir sie allerdings - zumindest für den Zeitpunkt des Routineaufrufs - wieder einschalten.

Wenn wir - was wohl meistens der Fall sein wird - Kombinationen von BASIC- und Assemblersprache verwenden, können wir den gesamten BASIC-Speicher bis $A000 frei halten, können auch den bei allen Beispielprogrammen so beliebten Bereich $C000 bis $D000 leer lassen und packen unsere Routinen weitgehend unter die ROMs, die dann jeweils beim Aufruf abgeschaltet werden. So haben wir eine Menge zusätzlichen Speicherplatz ergattert.

Nun können wir auch den letzten Rest des bislang unklaren Programms aus der letzten Folge verstehen. Nachdem wir den Inhalt der Speicherstelle 1 auf den Stapel gerettet haben (Zeilen $02CE und $02D0), schreiben wir $35 in den Prozessorport:

```
02D1 LDA #35
02D3 STA 01
```

$35 ist binär 0011 0101. Die Bits 0 bis 2, auf die es uns in diesem Zusammenhang ankommt, bewirken nun das Ausschalten des Interpreters und des Betriebssystems. Die Ein- und Ausgabe-Bausteine bleiben aktiv. Im weiteren Programmverlauf lesen wir die Speicherinhalte ab $E000, wobei wir nun den RAM-Inhalt erfassen.

Das sollte vielleicht noch mal klargestellt werden: Jedes Hineinschreiben in die mehrfach belegten Speicherbereiche (dabei sind die Ein- und Ausgabe-Bausteine aber ausgenommen) wird automatisch in den RAM-Bereich umgelenkt. Das ist ja auch klar: In ein ROM kann eben nicht geschrieben werden. Deshalb braucht man dabei die ROMs nicht auszuschalten.

Jeder Lesevorgang greift aber auf die ROMs zu, weshalb man sie in unserem Fall ausschalten muss. Wie schon oben beim Stapel erklärt, schalten wir durch das Zurückholen des vorher dorthin geretteten alten Inhalts der Speicherstelle 1 in den Prozessorport wieder den Normalzustand ein.

Die wandernden toten Briefkästen der Assembler-Alchemisten

Wir werden nun die beiden letzten noch ausstehenden Arten der Adressierung kennen lernen. Beides sind sogenannte indirekte Adressierungsarten.

Mit dem indirekten JMP-Befehl (zum Beispiel `JMP(0300)`) sind wir in der letzten Folge schon vertraut geworden. Wir hatten auch gelernt, dass es sich hierbei um einen absoluten Einzelgänger handelt, der nur für solch einen Sprung erlaubt ist.

Ebenso haben wir die indizierte Adressierung zu beherrschen gelernt: Das war die Sache mit den Indexregistern X oder Y. Eine Kombination aus beiden (also der indirekten und der indizierten) Adressierungsarten sind die indiziert-indirekte und die indirekt-indizierte Adressierung.

Die indirekt-indizierte Adressierung

Fangen wir mit der sehr häufig benutzten indirekt-indizierten Adressierung an: Man nennt sie auch »indirekt Y« oder »nachindizierte indirekte« Adressierung. Am Besten sehen wir

uns mal so einen Befehl an: LDA (FA),Y
Die Klammer erinnert uns an den indirekten JMP-Befehl. Tatsächlich hat sie hier auch dieselbe Funktion: In FA und FB steht ein Zeiger auf eine Adresse. Nehmen wir mal an, die Belegung der Speicher wäre:

```
FA 01
FB 80
```

und im Y-Register stünde eine 5. Der Zeiger FA/FB weist also auf die Speicherstelle 8001. Da haben wir also wieder das Prinzip des toten Briefkastens. Der Computer guckt in den hohlen Baum FA/FB (LSB in FA, MSB in FB) und findet dort die Treffpunktadresse. Nun sind diese toten Briefkästen aber auch den gegnerischen Alchemisten-Agenten bekannt. Es kommt also noch ein Trick dazu: Zur dort aufgefundenen Adresse wird der Inhalt des Y-Registers addiert. In unserem Fall fanden wir also in FA/FB die Adresse 8001, im Y-Register steht eine 5. Somit ist die endgültige Adresse 8001 + 5 = 8006.

Unser Beispiel »LDA (FA),Y« bewirkt daher, dass in den Akku der Inhalt der Speicherstelle 8006 geladen wird. Nachindiziert nennen manche die Adressierung, weil zunächst dem Zeiger nachgegangen wird, der in unserem Beispiel auf 8001 weist, und erst danach durch Addition des Inhalts des Y-Registers die endgültige Speicherstelle (hier also 8006) berechnet wird.

Als Zeiger (also die Adresse in der Klammer) sind nur Zeropagespeicherstellen verwendbar, als Indexregister darf man hier nur das Y-Register gebrauchen. Von den bisher behandelten Befehlen können ADC, CMP, LDA, SBC und STA mit dieser Adressierungsart verwendet werden. Genaueres finden Sie wieder in der Tabelle mit der Befehlsübersicht (Tabelle 16). Bevor wir uns

Tab. 16: Übersicht der in dieser Folge vorgestellten Befehle

Befehlswort	Adressierung	Byte-Anzahl	Kode		Dauer in Taktzyklen	Beeinflussung von Flaggen
			HEX	DEZ		
LDA	indirekt, X	2	A1	161	6	N, Z
	indirekt, Y	2	B1	177	5*	N, Z
STA	indirekt, X	2	81	129	6	-
	indirekt, Y	2	91	145	6	-
ADC	indirekt, X	2	61	97	6	N, V, Z, C
	indirekt, Y	2	71	113	5*	N, V, Z, C
SBC	indirekt, X	2	E1	225	6	N, V, Z, C
	indirekt, Y	2	F1	241	5*	N, V, Z, C
CMP	indirekt, X	2	C1	193	6	N, Z, C
	indirekt, Y	2	D1	209	5*	N, Z, C
PHA	implizit	1	48	72	3	-
PLA	implizit	1	68	104	4	N, Z
PHP	implizit	1	08	8	3	-
PLP	implizit	1	28	40	4	alle
TSX	implizit	1	BA	186	2	N, Z
TXS	implizit	1	9A	154	2	-

* Wenn bei der Befehlsausführung eine Page-Grenze überschritten wird, muss noch ein Taktzyklus addiert werden.

dem anderen indirekten Adress-Modus zuwenden, wollen wir uns überlegen, wozu man die indirekt-indizierte Adressierung verwendet. Wie Sie sich natürlich erinnern können, konnte man mit der normalen indizierten Adressierung, zum Beispiel mit

```
LDA 8000,Y
```

durch Variation des Indexregisters (hier das Y-Register) 256 Speicherstellen erfassen (Y von FF herunter bis 00). Will man mehr als diese 256 berücksichtigen, dann muss eine neue Basis (im Beispiel also an Stelle der 8000) gewählt werden.

Um das zu illustrieren, sehen wir uns mal den Anfang eines Programms an, welches den gesamten Bildschirminhalt ausliest und nach E000 schreibt:

```
1000 LDY #00
1002 LDA 0400,Y
1005 STA E000,Y
1008 LDA 0500,Y
100B STA E100,Y
100E LDA 0600,Y
1011 STA E200,Y
1014 LDA 0700,Y
1017 STA E300,Y
101A DEY
101B BNE 1002
...
```

Wie Sie sehen, erfordert das durch die Tatsache, dass vier Blöcke zu je 256 Bytes übertragen werden müssen, immerhin schon 28 Bytes Programmtext.

Nun soll die indirekt-indizierte Adressierung verwendet werden, um dieselbe Aufgabe zu lösen.

Wir legen zunächst zwei Zeiger auf der Zeropage fest: FA/FB sollen die Bildschirmadresse enthalten, FC/FD die Zieladresse ab E000.

```
1000 LDA #00
1002 STA FA
1004 STA FC
```

Das waren die LSBs der Zeiger. Es folgen die MSBs:

```
1006 LDA #04
1008 STA FB
100A LDA #E0
100C STA FD
```

Damit sind die Zeiger festgelegt. Es sind vier Blöcke zu je 256 Bytes zu übertragen. Diese Blockanzahl legen wir als Zähler ins X-Register:

```
100E LDX #04
```

Dann laden wir ins Y-Register ebenfalls einen Zähler (den Index):

```
1010 LDY #00
```

Jetzt kann die eigentliche Übertragungsschleife starten:

```
1012 LDA (FA),Y
1014 STA (FC),Y
1016 DEY
1017 BNE 1012
```

Wenn das Y-Register wieder bei 0 angekommen ist (von der ersten 0 nach einem Unterlauf - siehe dazu Folge 3 - über FF, FE und so weiter bis 0), ist der erste Block übertragen. Wir erhöhen nun das MSB beider Zeiger um 1:

```
1019 INC FB
101B INC FD
```

Außerdem zählen wir den Blockzähler um 1 herunter:

```
101D DEX
101E BNE 1012
```

Wenn das Programm auf diese Weise auch drei Bytes mehr Speicherplatz braucht, ist doch leicht der Vorteil zu sehen: Müssen wir

nämlich (statt nur vier) mehr Blöcke übertragen (bis zu 255), dann verändert sich unser zweites Programm um keinen Deut (außer dem Zähler im X-Register, der nun mit der jeweils anderen Block-Anzahl geladen wird), während die erste Programmiertechnik für jeden weiteren Block um sechs Bytes erweitert werden muss. Es gibt noch eine ganze Reihe von Anwendungsmöglichkeiten, die die indirekt-indizierte Adressierung so attraktiv machen. Für Geschwindigkeitsfanatiker (ich selbst bin bei Grafik-Fragen auch einer!) muss aber gesagt werden, dass dem Speicherplatzvorteil ein Geschwindigkeitsnachteil gegenübersteht.
Jeder indirekt-indiziert adressierte Befehl braucht einen Taktzyklus länger als der vergleichbare absolut-indizierte Befehl. Zu diesen Feinheiten werden wir aber in späteren Folgen noch kommen.

Die indiziert-indirekte Adressierung

Wenden wir uns nun der letzten noch fehlenden Adressierungsart zu: der indiziert-indirekten. Man nennt sie auch »vorindizierte Indirekte« oder »Indirekt X«-Adressierung. Sehen wir uns zunächst ein Beispiel an:

```
STA (FA,X)
```

Auch hier drückt die Klammer wieder aus, dass der Klammerinhalt ein Zeiger ist. Das ist jetzt aber nicht das Bytepaar FA / FB, sondern zur angegebenen Adresse FA soll noch der Inhalt des X-Registers addiert werden.

Nehmen wir mal an, dort stünde eine 2, dann wird der Zeiger FC / FD mit diesem Befehl angesprochen, denn FA + 2 = FC und entsprechend FB + 2 = FD. Wenn die Speicherstellen FA bis FF folgenden Inhalt haben:

```
00FA 00
00FB 04        FA/FB = 0400
00FC 00
00FD E0        FC/FD = E000
00FE 10
00FF 80        FE/FF = 8010
```

dann könnte das eine ganze Tabelle von Zeigern sein, die jeweils durch den X-Registerinhalt angesprochen werden. Der Akkuinhalt wird in unserem Beispiel nach 0400 geschrieben, wenn im X-Register 0 steht, nach E000, wenn das X-Register eine 1 enthält und nach 8010, wenn statt dessen eine 2 im X-Register zu finden ist.

Sie werden sich vielleicht auch bei diesem Beispiel gefragt haben, was passiert, wenn im X-Register unseres Beispiels eine 3 steht. Nun, unser 8-Bit-Prozessor läuft über, und wir finden einen Zeiger 00/01.

Rein theoretisch ist diese Adressierungsweise ganz interessant. Aber auf der Zeropage ist's reichlich eng, und nur selten kommt man daher in die Lage, dort eine Zeigertabelle einzurichten, die man mittels des X-Registerinhalts und der indiziert-indirekten Adressierung abgreifen kann.
Die Bedeutung dieser Adressierungsart ist also nur recht gering. Außerdem erfordert sie sechs Taktzyklen zur Bearbeitung und ist somit auch noch recht langsam. Von den bisher bekannten Befehlen sind die Folgenden damit verwendbar: ADC, CMP, LDA und STA.

Bevor wir die Adressierung zu den Akten legen, sei noch erwähnt, dass manche Lehrbücher noch eine weitere Art, die Akkumulator-Adressierung, unterscheiden. Betroffen sind davon vier 1-Byte-Befehle, die wir noch kennen lernen werden und die man ebenso gut als implizit adressiert ansehen kann.

Die ersten Kernal-Routinen

Sicher werden Sie alle schon von der Kernal-Routine FFD2 gehört haben und sie vielleicht auch schon verwenden. Wenn nicht, um so besser, denn dann sind Sie noch nicht vom einseitigen Gebrauch dieses Instruments verdorben.

Die meisten Kernal-Adressen sind nämlich sehr vielseitig verwendbar, je nach den Vorgaben. Das ist wie mit einem Haushaltsgerät, das immer nur um Rühren von Ku-

chenteig eingesetzt wird. Dabei kann man damit auch noch Saft machen, Gurken schnitzeln, Getränke mixen ...

Genauso wie man in diesem etwas schiefen Vergleich die Gebrauchsanleitung kennen sollte, um die ganzen anderen Funktionen ausnutzen zu können, muss man hier noch einige Dinge über die Kernal-Aufrufe beherzigen.

Für jede Verwendung der Kernal-Sprungtabelle sollte man sich angewöhnen, dies in drei Schritten zu tun:

- die nötigen Vorbereitungen treffen
- Routineaufruf
- Fehlerabfrage und -behandlung

Fangen wir mit dem Punkt »Vorbereitungen« an. Einige Routinen brauchen Informationen, die ihnen erst durch andere Kernal-Routinen beschafft werden. Ruft man diese anderen Routinen vorher nicht auf, dann funktioniert auch der erwünschte Aufruf nicht richtig.

Wenn die Routine einen bestimmten Wert im Y-Register erwartet, dann muss der dort auch stehen. Wenn nicht, dann geht das Programm in die Hose. Bei jeder Kernal-Routine, die hier beschrieben wird, gebe ich alle nötigen Vorbereitungen an.

Der Routinenaufruf sollte immer mittels JSR erfolgen. Alle auf diese Weise aus der Kernal-Sprungtabelle abzurufenden Programme enden nämlich mit einem RTS. Damit keine wichtigen Werte aus dem Aufrufprogramm überschrieben werden, man sie also vor dem Aufruf der Kernal-Routine irgendwohin retten kann, gebe ich auch noch an, welche Register durch die Routine verändert werden und wieviel Stapelspeicherplatz bereitgehalten werden muss.

Die Routinen sind so konstruiert, dass beim Auftreten eines Fehlers nach der Rückkehr das Carry-Bit gesetzt ist. Durch Untersuchen des Carrys können so Fehler rechtzeitig erkannt und behandelt werden. Im Akku findet man in dem Fall eine Fehlernummer. Die Ausgabe der Fehlermeldung erfolgt also nicht - wie im BASIC - in Klarschrift. In Tabelle 17 sind die Fehlernummern und ihre Bedeutung aufgelistet. Welche Fehlernummern eine Routine ausgeben kann, wird ebenfalls von mir bei jeder Routinen-Besprechung angegeben.

Nummer	Text	Bedeutung
0	BREAK	Während des Programms wurde die RUN/STOP-Taste gedrückt.
1	TOO MANY FILES	Man kann maximal 10 offene Files einrichten.
2	FILE OPEN	Ein bereits geöffnetes File wurde nochmals geöffnet.
3	FILE NOT OPEN	Auf ein noch nicht geöffnetes File sollte zugegriffen werden.
4	FILE NOT FOUND	Das angeforderte File ist nicht verfügbar.
5	DEVICE NOT PRESENT	Das angesprochene Gerät zeigt keine Reaktion.
6	NOT INPUT FILE	Aus einem zum Schreiben geöffneten File kann nicht gelesen werden.
7	NOT OUTPUT FILE	In ein zum Lesen geöffnetes File kann nicht geschrieben werden.
8	MISSING FILE NAME	Bei Operationen, die einen Filenamen erfordern, fehlt dieser.
9	ILLEGAL DEVICE NUMBER	Das gegebene Kommando ist beim angesprochenen Gerät nicht möglich.

Tab. 17: Fehlernummern und ihre Bedeutung. Die Nummern findet man bei gesetztem Carry im Akku.

Nun aber zur ersten Routine FFD2, die wie ein Rattenschwanz eine Reihe weiterer nach sich zieht:

Name	**CHROUT**
Zweck	Ausgabe eines Zeichens
Adresse	$FFD2 (dez. 65490)
Vorbereitungen	(CHKOUT, OPEN)
	Zeichen im Akku
Fehler	0
Stapelbedarf	8
Register	Akku

Falls Sie diese Routine schon einmal benutzt haben, dann geschah es vermutlich ohne die Vorbereitungen CHKOUT und OPEN. Freundlicherweise hat unser Computer einige Voreinstellungen schon für uns getroffen. Denn normalerweise sendet CHROUT ein Zeichen über einen schon geöffneten Ausgabekanal, und der ist zum Bildschirm geschaltet. Ein kleines Beispielprogramm soll das illustrieren.

Zunächst laden Sie bitte den SMON ein und starten Sie ihn. Nun soll eine Texttabelle angelegt werden. Das funktioniert beim SMON am bequemsten über das K-Kommando. Geben Sie ein: K 6000. Der SMON antwortet mit:

```
'6000 ........................
```

Wenn Sie nun die <RUN/STOP>-Taste drücken, können Sie mit dem Cursor in diese Punktzeile fahren und einen Text schreiben:

```
'6000 HALLO ASSEMBLER-ALCHEMIST
.................
```

Sinnvoll - vor allem für die weitere Verwendung dieses Textes - ist es, ein »RETURN«, also dezimal 13 oder $0D anzuschließen. Dazu gibt es natürlich den Weg über den Assemblerbefehl. Zur Übung wollen wir aber das M-Kommando verwenden. Geben Sie ein (zuerst die <RETURN>-Taste betätigen) M 6018, dann wieder <RUN/STOP>, und fahren Sie mit dem Cursor auf Speicherstelle 601A (falls Sie in 6019 kein Leerzeichen $20 stehen haben, dann fügen Sie's jetzt noch ein). Geben Sie nun an Stelle des dort stehenden Bytes 0D ein, und drücken Sie die <RETURN>-Taste. Der Monitor sollte jetzt zeigen:

```
:6018 54 20 0D etc.
```

Unser Text soll mit einem BRK enden. Deshalb gehen wir jetzt mit dem SMON-Kommando A 601B in den Assembler-Modus und schreiben:

```
601B BRK
```

Nun folgt das eigentliche Programmchen, das Byte für Byte bis zur Null (BRK) den Text aus der gerade erstellten Texttabelle liest und mittels FFD2 auf den Bildschirm bringt:

```
601C LDY #00
601E LDA 6000,Y
6021 BEQ 602C
```

Das Y-Register wird als Index initialisiert, dann die Texttabelle in den Akku geladen. Wenn das Programm dabei auf die Null stößt, verzweigt es zum Ende. Jetzt folgt die Routine zur Bildschirmausgabe:

```
6023 JSR FFD2
6026 BCS 602D
```

Falls bei der Kernal-Routine etwas schief gelaufen ist, wird das Carry-Bit gesetzt, was wir überprüfen und zu einem BRK-Kommando verzweigen (das ist natürlich nur sinnvoll, solange ein Monitor oder Assembler aktiv ist). Nun erhöhen wir das Index-Register, und das Ganze beginnt von vorn:

```
6028 INY
6029 JMP 601E
602C RTS
602D BRK
```

Wenn wir nun aus dem SMON mit F und anschließendem X aussteigen und ein kleines BASIC-Aufrufprogramm machen (Bei »OUT OF MEMORY ERROR« bitte »NEW« eingeben):

```
10 PRINT CHR$(147)
20 SYS 24604 : REM = $601C
30 END
```

dann können wir uns die Wirkung unseres Programms ansehen: Nach »RUN« wird der Bildschirm gelöscht und unser Text ausgedruckt.

FFD2 nimmt uns also eine Menge Arbeit ab: Automatisch legt diese Routine in den Bildschirmspeicher den Bildschirmkode (sie rechnet also auch gleich ASCII, das wir ja eingegeben haben, in den POKE-Kode um) und in die dazugehörige Bildschirmfarbspeicherstelle den aktuellen Farbkode. Sie setzt außerdem noch den Cursor weiter.

Mit FFD2 kann man aber noch viel mehr machen! Schließlich ist ja der Bildschirm (Gerätenummer 3) nicht der einzige mögliche Empfänger.

Wir wollen als Nächstes mal eine Ausgabe mittels FFD2 auf den Drucker erzielen. Hier sind die Vorbereitungen allerdings nötig. Zunächst mal müssen wir uns noch zwei weitere Kernal-Routinen ansehen, nämlich CHKOUT und OPEN:

Name	**CHKOUT**
Zweck	Kanal zum Ausgang definieren
Adresse	$FFC9 (dez. 65481)
Vorbereitungen	OPEN, logische Filenummer ins X-Register
Fehler	0, 3, 5, 7
Stapelbedarf	4
Register	Akku, X-Register

Mit dieser Routine kann jedes File, das zuvor durch OPEN spezifiziert worden ist, zum Ausgabefile erklärt werden. Natürlich muss dann das derart angesprochene Gerät auch ein Ausgabegerät sein. Andernfalls ergibt sich ein Fehler. Bevor man Daten über einen Kanal senden will, muss CHKOUT durchgeführt werden.

Wenn die mittels OPEN übergebene Geräteadresse größer als 3 ist, sendet diese Routine automatisch auch ein LISTEN-Kommando an das Ausgabegerät. LISTEN setzt dann zum Beispiel den Drucker in Empfangsbereitschaft.

Die Durchführung von CHKOUT ist einfach (vorausgesetzt, man hat vorher OPEN aufgerufen): In das X-Register wird die logische Filenummer geschrieben und dann per `JSR FFC3 CHKOUT` angesteuert. Nun zur anderen Vorbereitung von FFD2, zu OPEN:

Name	**OPEN**
Zweck	Öffnen eines logischen Files
Adresse	$FFC0 (dez. 65472)
Vorbereitungen	SETLFS, SETNAM
Fehler	1, 2, 4, 5, 6
Register	Akku, X- und Y-Register

Die Routine OPEN an sich anzusprechen, ist relativ einfach. Es genügt ein `JSR FFC0`. Zuvor allerdings - der Rattenschwanz wird länger - muss mit SETNAM der Filename und mit SETLFS die logische Filenummer, die Geräteadresse und eventuell eine Sekundäradresse festgelegt sein. Erst danach kann das File durch OPEN geöffnet werden. Also sehen wir uns noch SETLFS und SETNAM an:

Name	**SETLFS**
Zweck	Spezifikationen eines logischen Files
Adresse	$FFBA (dez. 65466)
Vorbereitungen	logische Filenummer in Akku, Gerätenummer ins X-Register, Sekundäradresse ins Y-Register
Fehler	keine
Stapelbedarf	2
Register	keine

SETLFS legt für die anderen Kernal-Routinen logische Filenummer, Gerätenummer und Sekundäradresse fest. Die logische Filenummer ist dabei eine Schlüsselzahl, die in eine durch OPEN angelegte File-Tabelle weist. Die Gerätenummer kann zwischen 0 und 31 liegen. Dabei sind folgende Zuordnungen vorgesehen:

0 Tastatur
1 Datasette
2 RS232C-Kanal
3 Bildschirm

Gerätenummern ab 4 beziehen sich automatisch auf Geräte am seriellen Bus. Dabei gilt im Allgemeinen:

4 Drucker
8 Diskettenstation

Die Sekundäradresse ist eine Kommandonummer, die für das jeweils angesprochene Gerät spezifisch ist, zum Beispiel bewirkt »10« beim Drucker Commodore 1526, dass das Gerät in die Grundstellung geht (siehe jeweiliges Handbuch). Will man keine Sekundäradresse verwenden, dann muss FF ins Y-Register geschrieben werden.

Der Aufruf von SETLFS geschieht also in folgender Weise: In den Akku lädt man die gewünschte logische Filenummer, ins X-Register die Geräteadresse und ins Y-Register FF oder aber die Sekundäradresse. Danach erfolgt der Sprung mit JSR FFBA. Schließlich noch zu SETNAM:

Name	**SETNAM**
Zweck	Filenamen festlegen
Adresse	$FFBD (dez. 65469)
Vorbereitungen	Namenslänge in den Akku, LSB des Namenstextes ins X-Register, MSB des Namenstextes ins Y-Register
Fehler	keine
Stapelbedarf	2
Register	Akku, X- und Y-Register

Vor der Eröffnung eines Files mittels OPEN muss diese Routine den Filenamen festlegen. Dazu schreibt man in den Akku die Länge des Namens und in die Register X, Y die Startadresse (LSB ins X-Register, MSB ins Y-Register) der Namenstext-Tabelle. Der Ort dieser Tabelle ist frei wählbar. Wird kein Filename gewünscht, dann gibt man dem Akku die Lange 0 an. X- und Y-Register sind in dem Fall ohne Bedeutung.

Damit - sollte man meinen - hätten wir nun alle Bedingungen erfüllt, FFD2 zur Ausgabe auf den Drucker zu bewegen. Leider ist das noch nicht der Fall: FFD2 schließt nämlich das File und den Ausgabekanal nicht. Das kann - wenn man's nicht beachtet - zu Fehlern oder zur weiteren Ansprache des Druckers führen, auch wenn die gar nicht mehr erwünscht ist. Deswegen sollten noch zwei Kernal-Routinen angehängt werden, von denen die eine (CLRCHN) alle Ein- und Ausgabekanäle wieder in den Ausgangszustand zurückführt, und die andere (CLOSE) das File ordnungsgemäß schließt:

Name	**CLRCHN**
Zweck	Ein- und Ausgabekanäle in Ausgangsstellung bringen
Adresse	$FFCC (dez. 65484)
Vorbereitungen	keine
Fehler	keine
Stapelbedarf	9
Register	Akku, X-Register

Die Wirkung ist enorm: Mit einem Schlag werden alle Kanäle freigeräumt. Eingangskanälen wird ein UNTALK (dem Gerät wird gesagt: Halt den Mund), Ausgangskanälen ein UNLISTEN (das bedeutet soviel wie: Hör nicht mehr zu) übermittelt. Der Ausgangszustand wird wiederhergestellt: Tastatur als Eingabe-, Bildschirm als Ausgabegerät. Die endgültig letzte Routine für diesmal ist CLOSE:

Name	**CLOSE**
Zweck	Schließen logischer Files
Adresse	$FFC3 (dez. 65475)

Vorbereitungen logische Filenummer in Akku
Fehler 0
Stapelbedarf 2
Register Akku, X- und Y-Register

Wenn für ein File alle Ein- und Ausgabeoperationen beendet sind, kann es - nach Einschreiben der Filenummer in den Akku - mittels CLOSE ordnungsgemäß geschlossen werden. Der Eintrag in der Filetabelle wird auf diese Weise gelöscht.
So, jetzt sind wir so weit, dass wir die Textausgabe auf dem Drucker programmieren können. Abbildung 15 fasst die einzelnen Schritte noch mal zusammen.

Und hier das Programm dazu. Wir verwenden die im anderen Beispiel schon aufgebaute Texttabelle weiter. Zunächst also SETNAM:

```
601C LDA #00
601E JSR FFBD
6021 BCS 6053
```

Wenn ein Fehler aufgetreten ist, findet man ein gesetztes Carry-Bit.

In dem Fall wird zu einem BRK-Kommando verzweigt (was die Anwesenheit eines Monitors erforderlich macht, solange man sich noch nicht sicher ist, ob Fehlermeldungen auftauchen). Die Null im Akku besagt, dass kein Filename gewünscht ist. Dann kommt SETLFS:

```
6023 LDA #04
6025 LDX #04
6027 LDY #FF
6029 JSR FFBA
602C BCS 6053
```

Es wurde ein File festgelegt mit der logischen Filenummer 4, der Geräteadresse 4 und ohne Sekundäradresse. Jetzt geben wir das OPEN-Kommando:

```
602E JSR FFC0
6031 BCS 6053
```

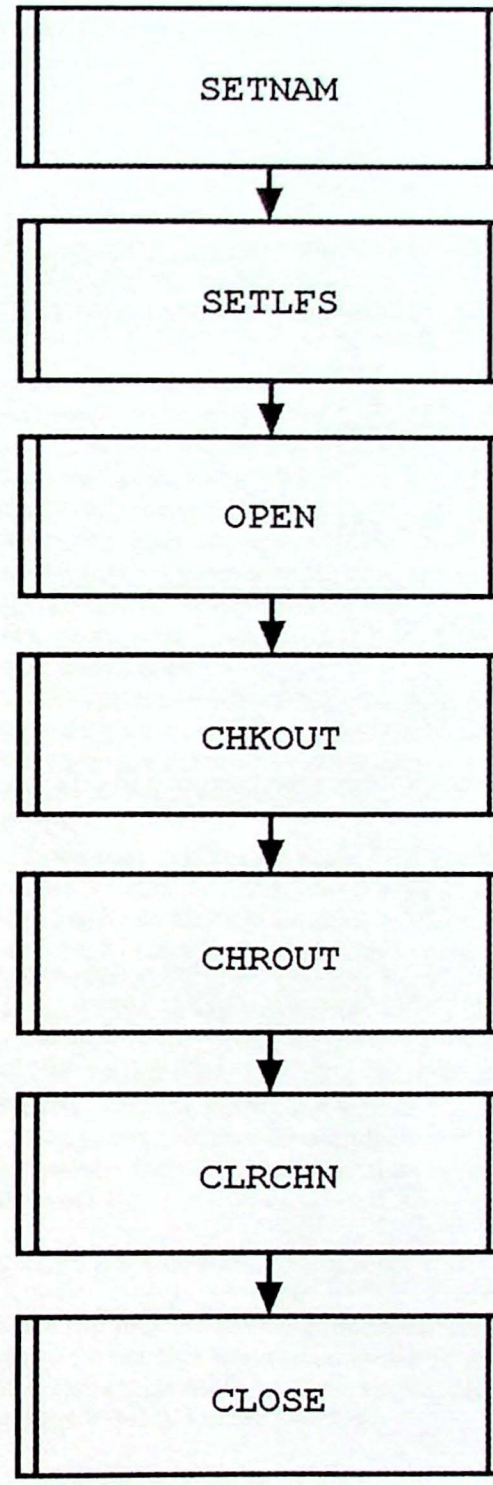


Abb. 15:
Die Abfolge der Routineaufrufe

Der Ausgabekanal wird mit CHKOUT definiert:

```
6033 LDX #04
6035 JSR FFC9
6038 BCS 6053
```

Damit sind alle Vorbereitungen erledigt, und die Zeichenausgabe kann wie im ersten Programm durchgeführt werden mit CHROUT:

```
603A LDY #00
603C LDA 6000,Y
603F BEQ 604A
6041 JSR FFD2
6044 BCS 6053
6046 INY
6047 JMP 603C
```

Alle Zeichen sind nun ausgedruckt. Wir rufen CLRCHN auf:

```
604A JSR
     FFCC
```

Als letzte Routine folgt nun noch CLOSE:

```
604D LDA #04
604F JSR FFC3
6052 RTS
```

Damit wurde das File Nummer 4 geschlossen. Anschließend erfolgte der Rücksprung aus dem Programm. Für die Fehlerbehandlung habe ich nur einen BRK vorgesehen, der sofortigen Registerüberblick erlaubt, wenn zum Beispiel der SMON im Speicher enthalten ist.

```
6053 BRK
```

Ohne Monitor im Speicher kann der Computer allerdings abstürzen oder im besten Fall einen BASIC-Warmstart durchführen. Wenn Sie so was also für Ihre Zwecke programmieren möchten, sollten Sie einen anderen Weg suchen, die Fehler aufzufangen. Man hat ja nicht immer einen Monitor eingeladen.

Mit diesen sieben Kernal-Routinen soll's für diesmal genug sein.

In der 64'er 12 / 1984 haben B. Schneider und K. Schramm in ihrer Serie »In die Geheimnisse der Floppy eingetaucht« gezeigt, wie man mittels der besprochenen und einiger anderer Routinen auch die Diskettenstation ansprechen oder sogar Floppy und Drucker zum »Spooling« veranlassen kann.
Ich habe das zwar schon öfter gesagt, muss es aber trotzdem immer wieder tun: Durch das Nachvollziehen fremder Programme kann man sehr viel lernen.

Oje, mein Versprechen, diesmal mit den Fließkommazahlen weiterzumachen, kann ich nicht halten. Auch unser Programmprojekt kommt nicht mehr dran. Beides hätte den Umfang dieser Folge mit Sicherheit gesprengt. Ich gebe Ihnen aber mein großes Ehrenwort, dass wir in der nächsten Ausgabe beide Themen weiterbehandeln.

Assembler ist keine Alchemie
Folge 8

Erschienen in: 64'er 4/1985
Verantwortlicher Redakteur: Georg Klinge
Autor: Heimo Ponnath

Irgendwann brauchen Sie sie bestimmt, die Rechnung mit Fließkommazahlen. Auch den Umgang mit logischen Ausdrücken wie AND, ORA oder EOR und wie man mit dem Schiebe-Befehl ASL multipliziert, sollen Sie beherrschen. Das alles erfahren Sie im achten Teil dieses Kurses.

Inzwischen wissen Sie ja, dass alle Daten im Computer im Binärformat enthalten sind. Nun soll noch mal erläutert werden, welcher Rechenweg der einfachste ist.

Als Beispiel nehmen wir die Zahl 1985. Man teilt diese Zahl so lange durch 2, bis das Ergebnis 0 wird. Jedes Mal notiert man sich den Rest, der entweder 0 oder 1 sein kann:

```
1985 : 2 = 992 Rest 1
 992 : 2 = 496 Rest 0
 496 : 2 = 248 Rest 0
 248 : 2 = 124 Rest 0
 124 : 2 =  62 Rest 0
  62 : 2 =  31 Rest 0
  31 : 2 =  15 Rest 1
  15 : 2 =   7 Rest 1
   7 : 2 =   3 Rest 1
   3 : 2 =   1 Rest 1
   1 : 2 =   0 Rest 1
```

Auch wenn Sie es noch nicht erkennen: Da steht schon das binäre Ergebnis. Von unten nach oben gelesen, ist das nämlich der Rest: 111 1100 0001

Nun reden wir ja von Fließkommazahlen. Also verändern wir unser Beispiel noch etwas. Jetzt soll uns die Zahl 1985,125 interessieren. In Folge 3 haben Sie gelernt, dass man das Komma verschieben kann, um daraus beispielsweise $1{,}985125 \times 10^3$ zu machen. Wir wollen uns das Verschieben des Kommas aber für etwas später aufheben und zunächst einmal außer dem schon umgewandelten Vorkommateil nun auch den Nachkommateil, also die 0,125, ins Binärformat übertragen.

Genauso, wie wir vorhin eine Kettendivision durch 2 verwendet haben, gebrauchen wir nun eine Kettenmultiplikation mit 2. Der gesamte Nachkommateil wird dabei verdoppelt.

Entweder ergibt sich dabei eine Vorkommastelle (das ist dann immer eine 1) oder das Ergebnis bleibt kleiner als 1. Wenn sich bei einem solchen Rechenschritt keine Vorkommastelle ergibt, schreibt man an die entsprechende Nachkommastelle der Binärzahl eine 0, andernfalls eine 1. Es wird so lange verdoppelt, bis keine Nachkommastellen mehr zur Verfügung stehen.

Das klingt ziemlich umständlich. Am Besten sehen Sie sich das jetzt mal an unserem Beispiel an:

```
0,125 x 2 = 0,250
```

Erste Nachkommastelle: 0. Beim ersten Verdoppeln hat sich keine neue Vorkommastelle ergeben, deshalb ist die erste Nachkommastelle der Binärzahl eine Null.

```
0,25 x 2 = 0,5
```

Zweite Nachkommastelle: 0. Auch beim zweiten Verdoppeln ermitteln wir keine neue Vorkommastelle, wodurch sich wieder eine Null als Nachkommastelle ergibt.

```
0,5 x 2 = 1,0
```

Dritte Nachkommastelle: 1. Hier hat sich nun eine Vorkommastelle beim Verdoppeln gebildet: Daher taucht als dritte Nachkommastelle unserer Binärzahl eine 1 auf. Gleichzeitig war das die letzte Nachkommastelle, denn unsere Ausgangszahl weist nach dem Komma nun nur noch eine Null auf.

Zur Übung wollen wir noch eine andere Zahl mit Nachkommastellen ins Binärformat überführen, nämlich 0,1.

```
0,1 x 2 = 0,2  1. Nachkommastelle: 0
0,2 x 2 = 0,4  2. Nachkommastelle: 0
0,4 x 2 = 0,8  3. Nachkommastelle: 0
0,8 x 2 = 1,6  4. Nachkommastelle: 1
```

Jetzt lässt man - das habe ich beim ersten Beispiel noch nicht erwähnt - diese neue Vorkommastelle einfach weg und rechnet wieder mit den Nachkommastellen weiter:

```
0,6 x 2 = 1,2  5. Nachkommastelle: 1
0,2 x 2 = 0,4  6. Nachkommastelle: 0
0,4 x 2 = 0,8  7. Nachkommastelle: 0
0,8 x 2 = 1,6  8. Nachkommastelle: 1
0,6 x 2 = 1,2  9. Nachkommastelle: 1
```

Das kommt Ihnen sicherlich von der 5. Verdoppelung her bekannt vor. Es zeigt sich, dass diese Rechnung nie aufgeht, weil sich eine periodische Zahl ergibt:

0,0001 1001 1001 100 ... Es kann Ihnen öfter bei der Zahlenumwandlung passieren, dass ein endlicher Dezimalbruch in einen unendlichen periodischen Binärbruch übergeht.

Kehren wir zurück zu unserem ersten Beispiel: 1985,125. Die ganze Umwandlung (Vorkomma- und Nachkommaanteil) führte zu: 111 1100 0001,001

Der dritte Schritt der Verwandlung von der Dezimalzahl zum Binärformat (nach 1. = Vorkommaanteil umwandeln, 2. = Nachkommaanteil umwandeln) ist das so genannte Normalisieren.

Das ist einfach das Verschieben des Kommas nach links (wie in unserem Beispiel) oder rechts, so lange, bis vor dem Komma nur noch Nullen stehen und direkt hinter ihm eine 1.

In Folge 6 haben wir gelernt, dass für jede Stelle, die das Komma nach links wandert, der Exponent um 1 höher wird. Unser Exponent ist im Moment noch Null (2^0 ist ja 1). Um also nach der Regel zu normalisieren, wird das Komma um 11 Stellen nach links verschoben. Der Exponent ist dann 1 (dez.) und unsere Zahl erscheint im neuen Gewand:

0.1111 1000 0010 01 E+1011. E+1011 heißt dabei Exponent und wird im Binärformat dargestellt (1011 (bin.) = 11 (dez.)). So weit, so gut. Alles bisher Unternommene hat Allgemeingültigkeit. Von nun an aber müssen wir uns spezialisieren auf den Commodore 64 (im VC 20 und einigen anderen Computern ist es aber auch so).

Der Exponent kann ja - je nachdem, ob das Komma zum Normalisieren nach links oder nach rechts verschoben wurde - positiv sein (wie bei unserem Beispiel), aber auch negativ.

Im Commodore 64 wird zum Exponenten die Zahl 128 addiert. Das ist dann Schritt 4. der im Beispiel zu 138 führt, womit wir schon das Exponentenbyte fertig haben: Exponent = dez. 139 = bin. 1000 1011 = hex. 8B. Hätten wir einen negativen Exponenten erhalten, zum Beispiel 20, dann stünde im Exponentenbyte nun dez. 108 beziehungsweise dasselbe im Binärformat.

Der Rest unserer Zahl, also die Mantisse, wird nun Schritt 5 unterzogen. Zunächst lässt man das Komma weg.

Die Binärzahl wird dann auf 4 Byte linksbündig aufgeteilt. In unserem Beispiel erhalten wir so:

```
1111 1000        0010 0100
 Byte 1           Byte 2

0000 0000        0000 0000
 Byte 3           Byte 4
```

Wie Sie sehen, werden die unbenutzten Bits mit Nullen aufgefüllt. Was nun noch nicht berücksichtigt wurde, ist das Vorzeichen der Mantisse. Es ist im Beispiel noch nicht zu erkennen, ob wir +1985,125 oder -1985,125 vorliegen haben. Das gehen wir nun im letzten Schritt (Nummer 6) an.

Im Commodore 64 gibt es zwei Möglichkeiten der Speicherung von Fließkommazahlen. Für Schritt 6 muss man sich entscheiden, wo man die Zahl haben will.

In Folge 6 ist schon mal der FAC erwähnt worden, der Fließkomma-Akkumulator 1, welcher die Speicherstellen dez. 97 bis 102 ($61 bis $66) belegt. Ein zweiter Fließkomma-Akkumulator, AFAC oder ARG genannt, belegt die Plätze dez. 105 bis 110 ($69 bis $6E). Diese Akkumulatoren haben für die Fließkommarechnungen eine ähnliche Bedeutung wie der Akku für die 1-Byte-Rechnungen.

Dort werden fast alle Ergebnisse abgelegt oder Zahlen abgerufen. Wir sehen, dass wir darin 6 Byte zur Verfügung haben. In Byte 97 liegt der Exponent in der von uns ermittelten Form. Byte 98 bis 101 sind die vier Mantissenbytes. Was ist in Byte 102? Das Vorzeichen!

Bit 7 dieses Bytes ist 0, wenn eine positive, und 1, wenn eine negative Zahl vorliegt. Das galt für den FAC, wie Sie aus den Speicherstellen schon gesehen haben. Für den ARG ist das aber ganz genauso. Sehen wir uns nun in Tabelle 18 unsere Beispielzahl im FAC und im ARG noch mal an.

Hier ist auch angedeutet, dass die restlichen 7 Bit (Bits 0 bis 6) des Vorzeichenbytes keine Rolle spielen. Sie werden später direkt in diese Akkumulatoren hineinsehen und allerlei Bit-Müll darin finden. Lediglich Bit 7 ist für uns von Bedeutung.

Eigentlich ist es ja eine ganz schöne Verschwendung, von einem Byte wie diesem Vorzeichenbyte lediglich ein einziges Bit zu nutzen. Wenn eine beliebige Fließkommazahl irgendwo im Computer abgespeichert wird, dann gilt ein anderes Format, das MFLPT-Format (von Memory FLoating PoinT).

	$	61	62	63	64	65	66
FAC	dez.	97	98	99	100	101	102
	$	69	6A	6B	6C	6D	6E
ARG	dez.	105	106	107	108	109	110
INHALT		1000 1011	1111 1000	0010 0100	0000 0000	0000	
BINÄR						0000	0 .. ..
HEX.		8B	F8	24	00	00	
DEZ.		139	248	36	0	0	
BYTE-Nr.		1	2	3	4	5	6
ERLÄUTERUNG		Exponent	MANTISSE				Vorzeichen

Tab. 18: So sieht die Zahl 1985,125 komplex in FAC und ARG aus.

Man speichert hier nur in 5 Bytes. Das Vorzeichenbyte fällt weg.

Wie aber merkt sich der Computer das Vorzeichen? Das ist ganz schlau eingefädelt: Es gibt nämlich in den 5 Byte (1 Exponentenbyte + 4 Mantissenbytes) ein überflüssiges Bit. Sie werden sich sicher erstaunt fragen, wo? Erinnern Sie sich doch bitte zurück an Schritt 3, das Normalisieren. Dort wurde so verfahren, dass rechts vom Komma eine 1 steht. Wenn da aber immer und ganz grundsätzlich diese 1 steht, dann muss man sie sich eigentlich gar nicht mehr besonders merken.

Man kann - vorausgesetzt, man berücksichtigt diese 1 im Bit 7 des ersten Mantissen-Bytes immer bei den Rechnungen - das Bit für andere Zwecke verwenden: Also als Vorzeichenbit. Taucht hier also eine 0 auf, dann liegt eine positive Zahl vor, ist es aber eine 1, dann signalisiert diese eine negative Zahl. Für das MFLPT-Format muss in unserem Beispiel also Bit 7 des ersten Mantissenbytes gelöscht werden (1985,125 ist ja nun mal positiv), und die komplette Zahl sieht im MFLPT-Format so aus:

1000 1011	0111 1000	0010 0100	0000 0000	0000 0000
Byte 1	↑ Byte 2	Byte 3	Byte 4	Byte 5
Exponent	Mantisse			

Der Pfeil weist auf das Vorzeichenbit. Man spricht hier auch vom »gepackten« Format.

Damit das alles nun nicht nur graue Theorie bleibt und Sie auch aus eigenem Erleben diese Zahlenformate sehen können, wollen wir hier ein kleines Testprogramm ausprobieren. Es wird Ihnen auch später noch gute Dienste leisten, wenn Sie mal irgendwelche Zahlen in das FLPT- (also FAC oder ARG) oder ins MFLPT-Format umrechnen müssen. Zu Fuß ist das ja - wie Sie nun wissen - ganz schön haarig! Wie so oft, besteht auch dieses Programm aus einem BASIC-Teil, der die Benutzerführung übernimmt, und zwei kleinen Maschinenroutinen, die per USR-Vektor angesprungen werden. In diesen Assembler-Programmteilen sind zwei Interpreter-Routinen verborgen, die nützlich und daher erklärenswert sind. Als Programm 2 ist das BASIC-Aufrufprogramm abgedruckt.

Es fragt zunächst mal, ob der SMON eingeladen ist. Der wird nämlich aus dem Programm heraus angesprungen. Wird die Frage mit »J« beantwortet, dann zeigt sich ein kleines Menü, andernfalls ist das Programm beendet: Der SMON muss erst eingeladen werden. Das Menü bietet drei Optionen: Eine Zahl kann im FAC (Option 1), im ARG (Option 2) oder im MFLPT-Format ab Speicherstelle $6800 (Option 3) betrachtet werden.

Für Option 1 wird der USR-Vektor auf die Einsprungadresse des SMON gestellt und dann mittels USR-Kommando die Zahl Z in den FAC übergeben. Dann schaltet sich der SMON ein, der nun mittels des Kommandos `M 0061` den Inhalt des FAC als Hex-Zahlen zeigt.

Option 2 richtet zunächst den USR-Vektor auf ein kleines Assembler-Programm ab $6000, welches den FAC-Inhalt in den ARG schiebt, dann den USR-Vektor auf den SMON richtet und schließlich auch diesen einschaltet. Auch hier wird mit dem M-Kommando dann per `M 0069` der ARG-Inhalt sichtbar.

Option 3 richtet den USR-Vektor auf eine Maschinenroutine, die bei $6100 beginnt. Dort wird der FAC-Inhalt nach $6800 und die folgenden Speicherstellen verschoben, und zwar ins MFLPT-Format. Anschließend erfolgt dann wieder das Ausrichten des USR-Vektors auf den SMON und Anschalten des SMON, wo man durch `M 6800` den Inhalt ansehen kann. Folgende Vorgehensweise empfehle ich Ihnen:

- Einladen des SMON
- Eintippen der beiden kleinen Assembler-Routinen mithilfe des SMON und Abspeichern (man kann einfach mit dem SMON-Kommando `S "Programmname",6000,610A` speichern).

- Wenn Sie die beiden Routinen schon gespeichert vorliegen haben, dann laden Sie sie jetzt ein. Jedenfalls sollten Sie nach dem Laden beider Assembler-Programme (SMON und die beiden Routinen) ein »NEW« eingeben, so dass alle Zeiger zurückgestellt werden.
- Erst jetzt Laden oder Eintippen des BASIC-Aufrufprogramms

Wenn Sie nun das Testprogramm starten und zum Beispiel unsere Zahl 1985,125 eingeben, werden Sie Folgendes finden:

Option 1: M 0061
:0061 8B F8 24 00 00 78 00 00

Option 2: M 0069
:0069 8B F8 24 00 00 78 D4 CE

Option 3: M 6800
:6800 8B 78 24 00 00 FF FF FF

Die Bytes, welche zu unserer Zahl gehören, sind unterstrichen. Sie können jeweils nach <RUN/STOP> noch mit dem SMON-Kommando $8B (oder einer anderen Sie interessierende Hexzahl) eine Ausgabe im Binär- und im Dezimalformat erreichen. So, nun aber endlich zu den beiden Assembler-Routinen.

Zur Option 2 gehört das folgende Programm, das bei $6000 beginnt:

Progr. 2: Testprogramm für die beiden kleinen Assembler-Routinen. Die Bedienung wird im Text erklärt.

```
5 REM *** TEST FUER FLPT UND MFLPT ***
10 POKE 52,96 : POKE 56,96 : CLR : PRINT CHR$(147)CHR$(17)CHR$(17)
15 PRINT "IST DER SMON EINGELADEN?" : INPUT "J/N";A$ : IF A$ = "N" THEN END
20 PRINT CHR$(17)CHR$(17)"{3 SPACE}FLPT IN FAC"TAB(25)"1" : PRINT
30 PRINT "{3 SPACE}FLPT IN ARG"TAB(25)"2" : PRINT
40 PRINT "{3 SPACE}MFLPT AB $6800"TAB(25)"3" : PRINT : PRINT
50 GET A$ : IF A$ < "1" OR A$ > "3" THEN 50
60 PRINT "AUSWAHL",A$ : PRINT : PRINT : INPUT "GEBEN SIE EINE ZAHL EIN";Z
65 PRINT CHR$(147)
70 ON VAL(A$) GOTO 100,200,300
100 REM *** FAC ***
110 POKE 785,0 : POKE 786,192 : REM *** USR-VEKTOR AUF SMON = $C000 ***
120 A=USR(Z)
200 REM *** ARG ***
210 POKE 785,0 : POKE 786,96 : REM *** USR-VEKTOR AUF $6000 ***
220 A=USR(Z)
300 REM *** MFLPT ***
310 POKE 785,0 : POKE 786,97 : REM *** USR-VEKTOR AUF $6100 ***
320 A=USR(Z)
400 REM ********************************
410 REM NACH MELDUNG DES SMON MIT DEN
420 REM KOMMANDOS
430 REM (1)    M 0061
440 REM (2)    M 0069
450 REM (3)    M 6800
460 REM DEN MONITOR EINSCHALTEN. DIE
470 REM EINGEGEBENE ZAHL IST DANN ALS
480 REM HEX-BYTES SICHTBAR.
490 REM ********************************
```

```
6000 JSR BC0C
```

$BC0C ist die erste Interpreter-Routine, die wir uns zu Nutze machen. Sie schiebt den Inhalt vom FAC in den ARG. Mehr dazu später.

```
6003 LDA #00
6005 STA 0311
6008 LDA #C0
600A STA 0312
```

Damit haben wir den USR-Vektor auf $C000 gestellt.

```
600D JMP C000
```

Das war das Einschalten des SMON. Im Grunde genommen könnten wir uns das Stellen des USR-Vektors ersparen. Es ist aber sinnvoll - vor allem bei langen Programmen -, wenn verstellte Vektoren nach Beendigung des Programms auf einem definierten Wert stehen. Nun noch die Routine für Option 3:

```
6100 LDX #00
6102 LDY #68
6104 JSR BBD4
```

Auch das ist wieder eine Interpreter-Routine: Sie schiebt den FAC-Inhalt in einen Speicherbereich, dessen Startbyte durch die beiden Index-Register angegeben wird (X-Register für LSB, Y-Register für MSB, hier also 6800). Dabei wird die Zahl vom FLPT-Format in das MFLPT-Format umgewandelt.

Das Prgrämmchen schließen wir ab mit einem Sprung zum Rest der ersten Routine:

```
6107 JMP 6003
```

Sehen Sie sich mal einige Zahlen im Fließkomma-Format an. Fast alle Operationen mit Zahlen vollführt unser Computer mit diesen Fließkommazahlen. Das ist dann beispielsweise der Grund dafür, dass aus einer BASIC-Zeile wie der folgenden: `IF INT (X*10)=INT(ABS(X*10)) THEN ...` auch bei positiven X-Werten (wo man mathematisch Gleichheit feststellt) manchmal die Bedingung als nicht erfüllt erkannt wird.

X wird sofort als Fließkommazahl in den FAC gelegt, mit einer Fließkomma-Zehn multipliziert, der ABS-Wert wird ebenfalls per Fließkomma-Arithmetik ermittelt und so weiter. Dabei treten häufig Rundungsprobleme auf, wenn ein Zwischenergebnis mehr als 32 signifikante binäre Nachkommastellen aufweist (wie wir es ja zum Beispiel beim periodischen Binärbruch gesehen haben, der sich aus der simplen Dezimalzahl 0,1 ergibt). Das Rechnen mit Fließkommazahlen im Computer öffnet zwar einen ungeheuren Zahlenraum für unsere Anwendungen, es geht aber viel langsamer als die 2-Byte-Arithmetik.

Immerhin müssen hier jedes Mal 6 Byte (beziehungsweise 5 bei MFLPT) berücksichtigt werden. Ich glaube aber kaum, dass wir jemals in die Verlegenheit kommen werden, beispielsweise eine Fließkomma-Addition programmieren zu müssen.

Eben weil unser C 64 fast alle Zahlenoperationen mit Fließkomma-Formaten durchführt, sind nahezu alle Eventualitäten schon als fertige abrufbare Programme im Interpreter enthalten. Wir müssen nur wissen, wie unsere Zahlen aussehen (das haben Sie ja nun gelernt), wo und wie man sie für Operationen bereithält und wo und wie man die entsprechenden Routinen finden kann. Einen der wichtigsten Wege, unsere Zahlen ans Maschinenprogramm zu übergeben, haben Sie schon kennen gelernt: Das Argument der USR-Funktion landet automatisch im FLPT-Format im FAC.

Die beiden ersten Interpreter-Routinen

Von nun an sollen nach und nach Interpreter-Routinen vorgestellt werden. Das ist allerdings nicht so einfach wie bei der Kernal-Sprungtabelle. Es gibt für die Letzteren viele recht gut dokumentierte Listen. Für die In-

terpreter-Routinen ist kaum Literatur vorhanden. Will man ähnlich erfassen wie die Kernal-Routinen, dann muss man ROM-Listings wälzen und vor allem probieren, probieren, probieren ... Nun zur ersten, schon verwendeten Routine:

Name	**MOVAF**
Zweck	Übertragen des FAC in den ARG
Adresse	$BC0C (dez. 48140)
Vorbereitungen	Wert in FAC
Speicherstellen	$61 - 66 (FAC), $69 - 6E (ARG), $6F, $70
Stapelbedarf	4
Register	Akku, X-Register

Diese Routine ist deswegen so wichtig, weil viele Rechenoperationen, die zwei Zahlen verknüpfen, zwischen dem FAC und dem ARG abgewickelt werden. Wenn Sie unser kleines Testprogramm mal mit der Option 2 laufen lassen und hinterher nicht nur mit `M 0069` in den ARG, sondern auch mit `M 0061` in den FAC hineinsehen, dann stellen Sie fest, dass der FAC-Inhalt noch immer vorhanden ist. Allerdings ist das nicht immer der Fall. MOVAF rundet nämlich - wenn nötig - vorher noch den FAC-Inhalt, der dann natürlich anders aussieht.

Fast noch häufiger benutzt man die zweite Interpreter-Routine:

Name	**MOVMF**
Zweck	Übertragung von FAC in Speicher unter Umrechnung ins MFLPT-Format
Adresse	$BBD4 (dez. 48084)
Vorbereitungen	Wert in FAC, Zieladresse in X- und Y-Register (X = LSB, Y = MSB)
Speicherstellen	$61 - 66 (FAC), $70, $22, $23
Stapelbedarf	4
Register	Akku, X- und Y-Register

Außer den unter »Speicherstellen« genannten sind natürlich auch noch die Zieladresse und deren vier nachfolgende Bytes in die Routine einbezogen (das MFLPT-Format besteht ja aus 5 Byte). $22/23 ist ein für die Operation verwendeter Zeiger.

MOVMF wird häufig dann verwendet, wenn Werte, aus welchen Gründen auch immer, außerhalb der Fließkomma-Akkumulatoren gelagert werden müssen.

Es wird Ihnen vielleicht aufgefallen sein, dass im Gegensatz zur Beschreibung der Kernal-Routinen die Rubrik »Fehler« fehlt. Der Grund ist, dass es keine solchen Sicherungen bei den Interpreter-Routinen gibt.

Was passieren kann, ist unter bestimmten Bedingungen das Ansteuern von normalen BASIC-Fehlermeldungen, die aber nicht immer den tatsächlichen Zustand wiedergeben.

Wenn Ihnen mal bei der Programmierung mit Interpreter-Routinen Zweifel aufkommen, dann verfolgen Sie lieber den Programmweg mittels eines ROM-Listings und schalten Sie eigene Fehler-Routinen ein.

Das war aber nur für die Fortgeschrittenen gesagt. Wir werden uns erst nach und nach dahin vortasten. Zunächst fehlen uns ja noch ein paar Assembler-Kenntnisse. Mit dem nächsten Abschnitt soll das besser werden.

Assembler-Befehle zum Beherrschen von Bits

Bevor wir damit anfangen, muss ich Ihnen noch sagen, dass Sie sich Zeit lassen sollten. Das Pensum, das ich Ihnen heute zumute, ist wirklich ganz gewaltig.

Zunächst sehen wir uns die logischen Befehle `AND`, `ORA` und `EOR` an, dann Bit-Schiebereien mittels `ASL`.

`LSR`, `ROL` und `ROR` sollen uns dann in der nächsten Ausgabe beschäftigen.

Fangen wir also mit `AND` an. `AND` verknüpft den Akku-Inhalt Bit für Bit mit dem angegebenen Wert nach den Regeln der logischen UND-Verknüpfung.

Es gibt allerlei Adressiermöglichkeiten für diesenBefehl:

```
AND 6000      absolut
AND FE        Zeropage absolut
AND #07       unmittelbar
AND 6000,X    absolut-X-indiziert
AND 6000,Y    absolut-Y-indiziert
AND (FA,X)    indiziert-indirekt
AND (FB),Y    indirekt-indiziert
AND FE,X      Zeropage-absolut-X-
              indiziert
```

Damit haben wir eine ganze Menge an Möglichkeiten. Erinnern Sie sich noch an die Regeln einer UND-Verknüpfung? Wenn nicht, dann sehen Sie sich noch mal Tabelle 19 an:

Tab. 19: Wahrheitstabelle zur AND-Verknüpfung

AND	0	1
0	0	0
1	0	1

Sie erkennen, dass zwei miteinander AND-verknüpfte Bits nur dann als Ergebnis 1 haben, wenn in beiden Bits der Wert 1 steht. Man kann mittels AND ganz gezielt Bits löschen. Nehmen wir mal als Beispiel an, wir wollen geshiftete Zeichen (das sind die mit den Kodes größer als 128) in normale Zeichen umwandeln. Dazu bringen wir die Zeichenkodes in den Akku und löschen Bit 7. Übrig bleibt dann der Kode für das ungeshiftete Zeichen.

Für das Löschen von Bit 7 brauchen wir eine sogenannte UND-Maske, die dafür sorgt, dass alle anderen Bits unverändert bleiben. An den Stellen muss in dieser Maske also eine 1 stehen (denn 0 AND 1 ergibt 0, 1 AND 1 ergibt 1). Lediglich Bit 7 der Maske muss 0 sein. Die Maske muss also heißen:
0111 1111 - $7F - dez. 127

Nehmen wir an, im Akku befände sich der Kode für ein geshiftetes A, also dez. 193 (binär 1100 0001, $C1), dann ergibt die AND-Verknüpfung mit der Maske:

```
Akku          1100  0001  Shift A
Maske         0111  1111
AND           ----------
Jetzt im Akku: 0100 0001  norm. A
```

(Das normale A hat den Dezimal-Kode 65, $41). Man kann also, je nach Wahl der Maske, beliebige Bits löschen.

AND ist, je nach der gewählten Adressierungsart, ein 2- oder 3-Byte-Befehl. Weil das Ergebnis im Akku steht, können Flaggen beeinflusst werden. Die N- und die Z-Flagge reagieren auf das Ergebnis.

Im Gegensatz zu BASIC, wo es nur eine ODER-Verknüpfung gibt, nämlich OR, existieren im Assembler zwei davon. Man unterscheidet ein »inklusives« und ein »exklusives« ODER.

Die inklusive ODER-Verknüpfung des Akkus mit den angegebenen Daten geschieht mit dem Assembler-Befehl ORA. ORA entspricht dem BASIC-Befehl OR. Alle Adressierungsarten, die dem AND-Befehl offen stehen, können auch auf ORA angewendet werden. Wenn man Bits ORA-verknüpft, findet man folgende Ergebnisse:

```
0 ORA 0 = 0
0 ORA 1 = 1
1 ORA 0 = 1
1 ORA 1 = 1
```

Auch hier ist eine so genannte Wahrheitstabelle recht einprägsam (Tabelle 20):

ORA	0	1
0	0	1
1	1	1

Tab. 20: Wahrheitstabelle zur ORA-Verknüpfung

Während man mit AND gezielt Bits löschen kann, ist es mit ORA möglich, Bits zu setzen. Auch dazu verwendet man eine Maske, die an allen Stellen, an denen Bits unverändert bleiben sollen, eine 0, sonst aber eine 1 enthält. Nehmen wir noch mal das Beispiel von vorhin und wandeln nun das ungeshiftete Zeichen in ein geshiftetes um.

Wir müssen also Bit 7 wieder setzen: Da muss in der Maske dann eine 1 stehen. Alle anderen Bits bleiben unverändert, wenn die Maske dort eine Null aufweist. Die Maske muss daher heißen:

1000 0000 - $80 - dez. 128. Im Akku soll das ungeshiftete B stehen (Kode dez. 66, $42, bin. 0100 0010). Die Rechnung sieht dann so aus:

```
Akku          0100  0010  norm. B
Maske         1000  0000
ORA           ---------------
Jetzt im Akku: 1100  0010  Shift B
```

Je nach Art der Maske kann man also ein oder mehrere Bits setzen. Im Beispiel ist auch der Einfluss dieses Befehls auf die Flaggen zu erkennen. Der Akku-Inhalt vor der ORA-Operation hatte kein Bit 7, also keine gesetzte N-Flagge. Danach ist Bit 7 gesetzt und die N-Flagge zeigt eine 1. Außer der N-Flagge kann - ebenso wie beim AND-Befehl - auch noch die Z-Flagge reagieren. ORA ist je nach Adressierungsart ein 2- oder 3-Byte-Befehl.

Während zwei Bits in der ORA-Verknüpfung eine 1 ergeben, wenn beide oder eins von beiden gesetzt sind, schließt die EOR-Verknüpfung den ersten Fall aus. EOR ist die exklusive ODER-Verknüpfung. Sie lässt sich sprachlich erfassen im »entweder ... oder ...«, also beispielsweise: Beim Roulette fällt die Kugel entweder auf Rouge oder auf Noir, beides zusammen ist nicht möglich. Die Regeln bei EOR sind also:

```
0 EOR 0 = 0
0 EOR 1 = 1
1 EOR 0 = 1
1 EOR 1 = 0
```

Die Wahrheitstabelle dazu sieht wie folgt aus:

EOR	0	1
0	0	1
1	1	0

Tab. 21: Wahrheitstabelle zur EOR-Verknüpfung

Wozu verwendet man EOR? Es fällt Ihnen vielleicht auf, dass wir die aus BASIC bekannte NOT-Funktion nicht in Assembler vorliegen haben.

Obwohl EOR einige viel weitergehende Verwendungsmöglichkeiten aufweist als NOT (aber auf Boolesche Algebra wollen wir hier nicht eingehen), kann man es mit gleicher Wirkung einsetzen.

Wir haben beispielsweise in den ersten Folgen dieses Kurses negative Zahlen durch Komplementieren erzeugt. Dabei sollte jedes Bit in sein Gegenteil verkehrt werden. Das wäre die Aufgabe einer NOT-Funktion. Durch ein EOR FF können wir dasselbe erreichen. Sehen wir uns wieder ein Beispiel an. Im Akku steht dez. 15 ($0F, bin. 0000 1111):

```
Akku          0000  1111
Maske         1111  1111 = $FF
EOR           ---------------
Jetzt im Akku: 1111  0000
```

Der neue Akku-Inhalt ist das Einerkomplement von dez. 15.

Auch EOR kann alle Adressierungsarten verkraften, welche die beiden anderen logischen Assembler-Befehle erlauben. Je nach der gewählten Art liegt dann ein 2- oder 3-Byte-Befehl vor. Auch hier werden die Z- und die N-Flagge beeinflusst.

Das waren also die logischen Befehle. Leider ist hier nicht der geeignete Ort, die Vielseitigkeit, die damit möglich ist, deutlich zu machen. Wenn Sie sich dafür interessieren, sollten Sie mal etwas über Boolesche Algebra lesen oder eine Einführung in die mathematische Logik.

Um dieses Thema abzuschließen, soll noch erwähnt werden, dass der BASIC-Interpreter so eingerichtet ist, dass er immer dann, wenn die Richtigkeit einer Aussage zu überprüfen ist, mit -1 bei wahrer Aussage und mit 0 bei falscher Aussage antwortet. Auf diese Weise kommen diese merkwürdigen BASIC-Programmzeilen ins rechte Licht, in denen

Sequenzen auftauchen wie: C = A - 161 - 33 * (A < 255) - 64 * (A < 192) - 32 * (A < 160) + 32 * (A < 96) - 64 * (A < 64).

Jedes Mal, wenn zum Beispiel A < 64 ist, tritt an Stelle der Klammer ein -1 auf. Übrigens ist diese Formel eine schöne kurze Möglichkeit, ASCII-Kode (hier A als Variable) in Bildschirmkode umzurechnen (der Bildschirmkode steht dann in der Variablen C).

Kommen wir nun zur zweiten Gruppe von Assembler-Befehlen, die Bit-Manipulationen erlauben: den Verschiebe-Befehlen. Fangen wir dabei mit ASL an, was vom englischen »Arithmetic Shift Left« kommt. Zu deutsch heißt das dann »Arithmetisches nach links schieben«. Davon sind wir aber auch noch nicht schlauer. Sehen wir uns an, was dieser Befehl tut (Abbildung 16):

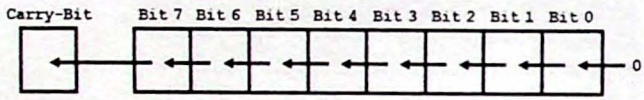


Abb. 16: Wirkung des ASL-Befehls - arithmetisches Linksschieben

Der gesamte Inhalt des Akkus beziehungsweise der Speicherstelle (je nach Adressierung) wird um eine Bit-Position nach links verschoben. Das vorherige Bit 7 wandert in die Carry-Flagge, alle anderen Bits erhalten eine um 1 höhere Position, das frei gewordene Bit 0 wird mit einer 0 aufgefüllt. Toll!

Aber was soll das? Zur Erklärung machen wir noch mal einen kurzen Ausflug zu unserem normalen dezimalen Zahlensystem.

Nehmen wir mal die Zahl 123. Bei der Einführung in die Fließkommazahlen hatten wir das Komma zu verschieben gelernt. 123 ist ja dasselbe wie 123,00. Wenn wir das Komma um eine Stelle nach rechts verschieben, erhalten wir 1230,0 (dabei lassen wir jetzt mal den Exponenten außer Acht, der wäre ja -1, weil $123{,}00 = 1230{,}0 * 10^{-1}$). Man kann das Ganze auch anders herum sehen:

Wir haben die Zahl 123 eine Stelle nach links verschoben und die frei gewordene Stelle ganz rechts mit einer Null aufgefüllt. 1230,0 ist das Zehnfache von 123,00. Die Verschiebung um eine Stelle nach links hat also zur Multiplikation unserer Zahl mit der Basis unseres Zahlensystems (also 10) geführt. Eine zweimalige Linksverschiebung führt zu 12300, dem 100-fachen Wert unserer Ausgangszahl. Wir haben also die Zahl 123,00 mal 10 mal 10 genommen, das sind 10^2. Jede Linksverschiebung erhöht unseren Ausgangswert um eine Zehnerpotenz oder - anders ausgedrückt - erhöht den Multiplikator um eine Zehnerpotenz und deshalb natürlich auch das Ergebnis (einmal linksschieben: Multiplikator = 10 = 10^1, zweimal linksschieben: Multiplikator: 100 = 10^2 und so weiter).

Im Binärsystem, zu dem wir nun wieder zurückkehren, ist die Zahlenbasis die Zahl 2. Einmal linksschieben entspricht dann einer Multiplikation mit $2^1 = 2$. Das zweimalige Linksschieben führt zur Multiplikation mit $2^2 = 4$ und so weiter. Nehmen wir als Beispiel die Zahl 3, welche im Binärsystem 0000 0011 heißt:

Bis jetzt landete im Carry-Bit immer eine Null. Wenn wir nun noch mal linksschieben, finden wir darin eine 1, die offensichtlich als Bit 8 unseres Ergebnisses dienen muss:

1. ASL führt zu 0000 0110 = dez. 6 ($2^1 * 3 = 2 * 3 = 6$)
2. ASL führt zu 0000 1100 = dez. 12 ($2 * 6 = 12$, $2^2 * 3 = 4 * 3 = 12$)
3. ASL führt zu 0001 1000 = dez. 24 ($2 * 12 = 24$, $2^3 * 3 = 8 * 3 = 24$)
4. ASL führt zu 0011 0000 = dez. 48 ($2 * 24 = 48$, $2^4 * 3 = 16 * 3 = 48$)
5. ASL führt zu 0110 0000 = dez. 96 ($2 * 48 = 96$, $2^5 * 3 = 32 * 3 = 96$)
6. ASL führt zu 1100 0000 = dez. 192 ($2 * 96 = 192$, $2^6 * 3 = 63 * 3 + 192$)
7. ASL(1) führt zu 1000 0000 = (mit Carry als Bit 8) dez. 384 ($2 * 192 = 384$, $27 * 3 = 128 * 3 = 384$

Daraus folgt, dass immer dann, wenn man sich nicht hundertprozentig sicher ist, eine Abfrage des Carry-Bits erfolgen sollte, sofern man ASL zum Rechnen einsetzt (BCC

beziehungsweise BCS bieten sich da an). Dazu kommen wir noch. Sehen wir uns zunächst mal an, wie ASL adressierbar ist:

```
ASL          ohne Adresse, der Akkuinhalt wird nach links verschoben. Manchmal als eigene Adressierungsart bezeichnet
ASL 6000     absolut
ASL FE       Zeropage-absolut
ASL 6000,X   absolut-X-indiziert
ASL FA,X     Zeropage-absolut-X-indiziert
```

Je nach Adressierung tritt ASL dann als 1-, 2- oder 3-Byte-Befehl auf. Die N-, die Z- und die Carry-Flagge werden beeinflusst. Das Ergebnis steht bei der ersten Adressierungsart (also ASL ohne Adresse) im Akku. In den anderen Fällen findet man es in der jeweiligen Speicherstelle.

Nun gut, werden Sie sagen, man kann also mittels ASL Zahlen mit 2, 4, 8, 16, 32 usw. multiplizieren. Was aber, wenn man mal 40 nehmen will? Da gibt es einige Möglichkeiten, die ein bisschen den Erfindungsgeist ansprechen. Man kann ja, wenn irgendeine Zahl Z mal 40 genommen werden soll, dafür schreiben: 40 * Z = (32 + 8) * Z = 32 * Z + 8 * Z

Schon haben wir wieder Multiplikatoren, die den Einsatz von ASL ermöglichen. Die beiden Zwischenergebnisse (als 32 * Z und 8 * Z) speichern wir irgendwo ab und zählen sie dann zusammen.

Wenn Z zum Beispiel 3 wäre, könnte man das so programmieren:

```
6000 STA 6100
```

Dabei sollte im Akku Z also die 3 stehen, die wir nun zwischengespeichert haben.

```
6003  ASL
6004  ASL
6005  ASL
6006  ASL
6007  ASL
```

Jetzt liegt im Akku der 32-fache Wert von 3, also 96, vor, und wir speichern dieses Zwischenergebnis ab.

```
6008 STA 6101
600B LDA 6100
```

Wir haben nun den Wert 3 aus dem Zwischenspeicher $6100 wieder in den Akku geholt und schieben ihn 3-mal nach links, um den 8-fachen Wert zu erhalten.

```
600E ASL
600F ASL
6010 ASL
```

Nun erfolgt das Zusammenzählen beider Zwischenergebnisse. Dabei ist ja 8 * Z noch im Akku.

```
6011 CLC
6012 ADC 6101
```

Damit ist die Aufgabe gelöst. Das Ergebnis steht im Akku und kann nun weiterverwendet werden.

Auf diese Weise kann man immer einen Multiplikator in eine Zweierpotenz (2, 4, 8, 16, ...) und weitere Summanden zerlegen. Dies ist allerdings eine zwar schnelle, aber doch recht eingeschränkte Art der Multiplikation. Außerdem haben Sie noch nicht erfahren, wohin man denn nun am Besten mit BCC verzweigt, wenn die 8 Bits des Ergebnisses überlaufen.

Das alles aber, liebe Leser, erfahren Sie erst beim nächsten Mal. Für heute war's lang und schwer genug! Wir werden dann einen eleganten Weg der 16-Bit-Multiplikation und Division erarbeiten. Und unser Programmprojekt soll natürlich endlich weiter gedeihen. Abschließend finden Sie in Tabelle 22 noch alles Wissenswerte zu den neuen Befehlen.

Tab. 22: Alles Wissenswerte zu den neuen Assembler-Befehlen

Befehlswort	Adressierung	Byte-Anzahl	Kode		Dauer in Taktzyklen	Beeinflussung von Flaggen
			HEX	DEZ		
AND	absolut	3	2D	45	4	N, Z
	0-page-absolut	2	25	37	3	N, Z
	unmittelbar	2	29	41	2	N, Z
	abs.-X-indiz.	3	3D	61	4*	N, Z
	abs.-Y-indiz.	3	39	57	4*	N, Z
	indiz.-ind.	2	21	33	6	N, Z
	ind.-indiz.	2	31	49	5*	N, Z
	0-page-X-indiz.	2	35	53	4	N, Z
ORA	absolut	3	0D	13	4	N, Z
	0-page-absolut	2	05	05	3	N, Z
	unmittelbar	2	09	09	2	N, Z
	abs.-X-indiz.	3	1D	29	4*	N, Z
	abs.-Y-indiz.	3	19	25	4*	N, Z
	indiz.-ind.	2	01	01	6	N, Z
	ind.-indiz.	2	11	17	5*	N, Z
	0-page-X-indiz.	2	15	21	4	N, Z
EOR	absolut	3	4D	77	4	N, Z
	0-page-absolut	2	45	69	3	N, Z
	unmittelbar	2	49	73	2	N, Z
	abs.-X-indiz.	3	5D	93	4*	N, Z
	abs.-Y-indiz.	3	59	89	4*	N, Z
	indiz.-ind.	2	41	65	6	N, Z
	ind.-indiz.	2	51	81	5*	N, Z
	0-page-X-indiz.	2	55	85	4	N, Z
ASL	»Akkumulator«	1	0A	10	2	N, Z, C
	absolut	3	0E	14	6	N, Z, C
	0-page-absolut	2	06	06	5	N, Z, C
	abs.-X-indiz.	3	1E	30	7	N, Z, C
	0-page-X-indiz.	2	16	22	6	N, Z, C

* Wenn bei der Befehlsausführung eine Page-Grenze überschritten wird, muss noch ein Taktzyklus addiert werden.

Assembler ist keine Alchemie
Folge 9

Erschienen in: 64'er 5/1985
Verantwortlicher Redakteur: Georg Klinge
Autor: Heimo Ponnath

Multiplizieren und Dividieren größerer Zahlen ist weder mit dem Taschenrechner noch in BASIC ein Problem. Mit Assembler sieht die Sache anfangs schon nicht mehr so einfach aus. Doch auch diese Hürde wird genommen. Einige Betriebssystemroutinen des C 64 nehmen uns dabei erhebliche Arbeit ab, man muss sie nur kennen.

Einige Versprechen sollen diesmal eingelöst werden: Die restlichen Bit-Verschiebe-Befehle werden Ihnen vorgestellt und auch gleich ein paar Anwendungen wie die 16-Bit-Multiplikation und die 16-Bit-Division. Außerdem soll endlich das Programmprojekt weitergeführt werden. Diesmal erzeugen wir einen Hilfsbildschirm und legen ihn abrufbereit unter den oberen ROM-Bereich. Bei der Gelegenheit lernen Sie auch gleich noch ein paar Interpreter-Routinen kennen.

Die restlichen Bit-Verschiebe-Operationen

Da wäre zunächst einmal das Gegenstück zu ASL. Den Befehl haben wir in der letzten Ausgabe kennen gelernt. Dort ging es ja um das Nach-Links-Schieben. Jetzt schieben wir nach rechts. LSR heißt der dazu nötige Befehl. Das kommt von »Logical Shift Right« und heißt zu deutsch »logisches Rechtsschieben«. Fragen Sie mich bitte nicht, weshalb »logisch«. Jedenfalls ist LSR ebenso für logische Bitprüfungen geeignet wie ASL.

Mittels LSR wird jedes Bit der adressierten Speicherstelle um einen Platz nach rechts geschoben. An die Stelle von Bit 7 tritt eine Null, und Bit 0 wandert in das Carry-Bit (siehe Abbildung 17).

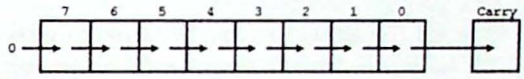


Abb. 17: Wirkung von LSR auf ein Byte

Erinnern Sie sich noch an das dezimale Linksschieben mit ASL aus der letzten Folge? Wir hatten festgestellt, dass jedes Linksschieben einer Dezimalzahl einer Multiplikation mit 10 entspricht.
Im umgekehrten Fall, also beim Rechtsschieben, muss jedes LSR einer Division durch 10 entsprechen:

25 000 wird durch LSR zu 2 500
2 500 wird durch LSR zu 250
250 wird durch LSR zu 25
und so weiter

Geht man von der Ausgangszahl (25 000) aus, dann ergibt sich der erste rechts verschobene Wert durch Division mit

$10^1 = 10$
der 2. durch $10^2 = 100$
der 3. durch $10^3 = 1000$
usw.

Es wird also durch Potenzen der Zahlenbasis 10 geteilt. Haben wir es - wie im Computer - mit Binärzahlen zu tun, deren Basis die 2 ist, dann teilen wir mit jedem LSR durch 2. Je nachdem, wie oft hintereinander das LSR auf eine Zahl ausgeübt wird, teilt man dann durch $2^1 = 2$, $2^2 = 4$, $2^3 = 8$ usw. Das konnte

man sich alles ja schon vorstellen, nachdem ASL zur Multiplikation verwendet wurde. Auch hier muss man immer das Carry-Bit abfragen, denn die Division kann ja unter Umständen nicht aufgehen, wie das folgende Beispiel der Division von 3 durch 2 zeigt: (3) 0000 0011 ergibt durch LSR: 0000 0001 und 1 im Carry-Bit. Das Ergebnis ist schon richtig, nämlich 1. Im Carry steht der Rest dieser Division, die 1. Weil der Rest für manche Berechnungen von Bedeutung ist, muss das Carry-Bit irgendwie erfasst werden. Wie man das erreicht, lernen wir später noch.

Leider ist diese Art der Division mittels LSR nicht so einfach verwendbar wie die Multiplikation mittels ASL. Während man dort durch geschicktes Aufteilen des Faktors ASL auch bei anderen Multiplikatoren als reine Zweierpotenzen anwenden konnte, ist das hier nicht so ohne Weiteres möglich. Bei Divisionen geht man deshalb lieber andere Wege. Die zeige ich Ihnen ebenfalls etwas später. LSR kann auf die gleiche Weise adressiert werden wie ASL:

```
LSR          auf den Akku bezogen
LSR 6000     absolut
LSR FE       Zeropage-absolut
LSR 6000,X   absolut-X-indiziert
LSR FA,X     Zeropage-absolut-X-indiziert
```

Im ersten Fall steht das Ergebnis im Akku, in den anderen Fällen in der jeweils adressierten Speicherstelle. Außer der N-Flagge, die in jedem Fall 0 wird, beeinflusst LSR auch die Carry-Flagge und unter Umständen die Z-Flagge. Je nach Adressierungsart liegt LSR als 1-Byte-, 2-Byte- oder 3-Byte-Befehl vor.

Sowohl bei ASL als auch bei LSR hatten wir festgestellt, dass man herausgeschobene Bits, falls sie noch von Bedeutung sind, irgendwie aus dem Carry-Bit (dort sind sie ja gelandet) an einen sinnvollen Ort schaffen muss. Das ist natürlich möglich: über eine Befehlskette, in der zunächst das Carry-Bit abgefragt wird. Hierzu ein Beispiel:

```
6000 BCC 6007
6002 LDA #01
6004 STA 8000
6007 etc.
```

Wenn das Carry-Bit frei ist, wird alles Weitere übersprungen. Wenn da drin etwas aufgetaucht ist, lädt man eine 1 (die ist ja im Carry-Bit) an die benötigte Speicherstelle (hier zum Beispiel 8000). Das kostet aber einige Bytes Speicherplatz und einige Taktzeiten Rechendauer. Außerdem erschwert sich die Programmierung, wenn man eine Zahl öfter verschiebt und dann alle Carry-Inhalte nach 8000 packen will. So kompliziert brauchen wir auch gar nicht zu arbeiten, denn unsere CPU kennt zwei Befehle, die das Bit-Verschieben und das Carry-Verschieben für uns erledigen. Das sind:

- ROL: ROtate Left (Linksrotieren)
- ROR: ROtate Right (Rechtsrotieren)

Sehen wir uns zunächst mal ROL (Abbildung 18) an:

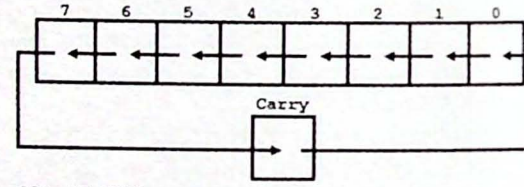


Abb. 18: Wirkung von ROL auf ein Byte

Wie bei ASL wandert jedes Bit um eine Position nach links. Das Bit 7 wird dabei in das Carry-Bit verschoben. In Bit 0 gelangt aber hier nicht eine 0 (wie bei ASL), sondern der Inhalt des Carry-Bits (wohlgemerkt der Inhalt, der dort war, bevor dort hinein Bit 7 geschoben wurde). Bevor wir auf den praktischen Nährwert dieses Befehls eingehen, sollen erstmal die Adressierungsmöglichkeiten aufgeführt werden:

```
ROL          auf den Akku bezogen
ROL 6000     absolut
ROL FE       Zeropage-absolut
```

```
ROL 6000,X  absolut-X-indiziert
ROL FE,X    Zeropage-absolut-X-
            indiziert
```

Je nach Adressierung kann es sich dann wieder um einen 1-Byte- bis 3-Byte-Befehl handeln. Die N-, Z- und natürlich die Carry-Flagge sind beeinflusst, und das Ergebnis des Befehls ist im Akku (erste Adressierungsart) oder in der angesprochenen Speicherstelle zu finden.

Wozu das Ganze? Abgesehen von der Möglichkeit, einzelne Bits auf diese Weise ohne Verlust aus einem Byte durch das Carry-Bit herausschieben zu können, um sie Prüfungen zu unterziehen, gibt es noch die Möglichkeit, einen Überlauf bei Rechenoperationen aufzufangen.

Erinnern Sie sich an die letzte Folge, wo wir mittels ASL Multiplikationen durchgeführt haben? Dort kann es unter gewissen Umständen ja leicht geschehen, dass ein Byte für das Ergebnis nicht mehr ausreicht.

Wir haben in den Beispielen schon die Überlegung durchgeführt, dass man mittels BCC oder BCS prüfen sollte, ob man eine signifikante Stelle (also eine führende 1) aus dem Byte herausgeschoben hat. Ist das der Fall, dann gibt es zwei Wege:

1. Man veranlasst den Ausdruck eines OVERFLOW ERROR, wenn nur 1-Byte-Zahlen zulässig sind.
2. Man schaltet um auf 2-Byte-Zahlen.

Sehen wir uns das mal an dem Schritt 1 des Beispiels aus der letzten Folge an. Dort hatten wir die Zahl 192 (binär 1100 0000) vorliegen (zum Beispiel in Speicherstelle 7000). Im Computer werden 2-Byte-Integers in der Form LSB / MSB verarbeitet. Wir schaffen also die Speicherstelle für das MSB von 192 in 7001.

Jetzt muss dort noch 0 drinstehen. Um bei nochmaliger Multiplikation mit 2 eine 16-Bit-Zahl als Ergebnis zu erhalten, verfährt man wie folgt:

```
6000 ASL 7000  Damit ist die führende 1
               ins Carry-Bit gewandert.
6003 BCC 6008  Das setzt man natürlich
               nur dann ein, wenn man
               nicht genau weiß, wel-
               ches Ergebnis zu erwar-
               ten ist.
               Wenn keine 1 ins Carry-
               Bit gelangte, kann man
               die nächste Zeile über-
               springen:
6005 ROL 7001  Damit wurde der Inhalt
               des Carry-Bits als Bit 0 ins
               MSB unseres Ergebnisses
               geschoben.
6008 etc.
```

Die Funktion dieser Befehlssequenz können Sie aus Abbildung 19 entnehmen. Diesem Befehl werden wir später bei der 16-Bit-Multiplikation und Division noch häufig begegnen.

Sehen wir uns nun noch den letzten der Bit-Verschiebebefehle an: ROR. In Abbildung 20 wird schematisch gezeigt, wie rotiert wird:

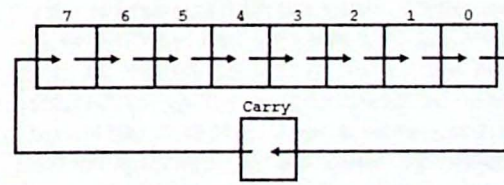


Abb. 20: Wirkung von ROR auf ein Byte

Jedes Bit wandert, wie bei LSR, um eine Stelle nach rechts. Als Bit 7 kommt (im Gegensatz zu LSR) der Inhalt des Carry-Bits herein. Bit 0 wird ins Carry-Bit geschoben. Adressiert werden kann ROR ebenso wie ROL:

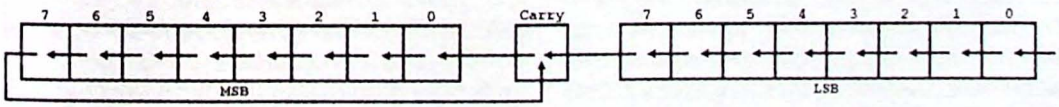


Abb. 19: Wirkung der Befehlskombination ASL(-) und ROL(-) auf zwei Bytes (LSB / MSB)

ROR	auf den Akku bezogen
ROR 6000	absolut
ROR FE	Zeropage-absolut
ROR 6000,X	absolut-X-indiziert
ROR FE,X	Zeropage-absolut-X-indiziert

Auch für die Byteanzahl, den Ort des Ergebnisses und die Flaggenbeanspruchung gilt dasselbe wie für ROL.

Die Einsatzmöglichkeiten für ROR sind allerdings geringer. Bei 16-Bit-Divisionen kann man zwar ROR einsetzen, um einen Unterlauf des MSB ins LSB aufzufangen. Weil man aber meist ohnehin andere Divisionsverfahren verwendet als das oben gezeigte mit LSR, erübrigt sich diese Anwendung in den meisten Fällen. Gut kann man ROR zu Bitprüfungen einsetzen. Das soll im nächsten Abschnitt an einem kleinen Beispiel gezeigt werden.

Zuvor aber noch eine Bemerkung: Wir sind nun durch den Befehlssatz des 6502-Assemblers fast hindurchgedrungen. Es fehlen uns nur noch - wenn ich mich nicht versehen habe - vier Befehle. Die allerdings hängen eng mit dem so genannten Interrupthandling zusammen, das uns wohl einige Zeit beschäftigen wird.

Schneller Joystick

In der 64'er 2 / 1985 veröffentlichte P. Siepen eine Routine zur Abfrage des Joystickports, die eine interessante Leserbrief-Reaktion hervorrief. M. Hartig sandte nämlich einen Verbesserungsvorschlag, in dem der uns interessierende Befehl ROR die Hauptrolle spielt. Bevor ich die allerdings vorstelle, muss erst noch geklärt werden, was und wie abgefragt wird.

Signale vom Joystick landen in den DATA-Ports A oder B des CIA 1. CIA heißt »Complex Interface Adapter« und ist die Institution unseres Computers, die den Verkehr mit der Außenwelt regelt. Wir haben zwei Stück davon (CIA 1 und CIA 2). Je nachdem, in welchen Port der Joystick gesteckt wurde, laufen die Signale in den Registern DC00 oder DC01 (dezimal 56320 oder 56321) ein. Wir nehmen im Weiteren DC00 an. Die Bits 0 bis 4 beziehen sich auf den Joystick:

Bit 0	oben
Bit 1	unten
Bit 2	links
Bit 3	rechts
Bit 4	Feuerknopf

Wenn keine dieser Möglichkeiten angesprochen ist, enthalten diese Bits den Wert 1. Drückt man beispielsweise den Feuerknopf, dann wechselt der Inhalt von Bit 4 zum Wert 0. Man muss also ständig diese Bits überprüfen und reagieren, sobald eines davon 0 wird. Die Lösung von P. Siepen, diese Abfrage in das Interruptprogramm einzubauen, ist sehr brauchbar. Dadurch hat der Computer die Möglichkeit, trotzdem an anderen Aufgaben weiterzuarbeiten.

Wir werden in den nächsten Folgen auf diese Programmiertechnik eingehen. Die Verbesserung von M. Hartig besteht darin, dass er nicht durch CMP-Befehle den Inhalt von DC00 prüft (was Zeit und auch Speicherplatz kostet), sondern mittels ROR Bit für Bit nach rechts in das Carry-Bit schiebt und dieses dann mit BCC abfragt. Sobald die Carry-Flagge nämlich frei ist, ist die zu dem Bit gehörige Joystickfunktion gefragt. Nun die Abfrageroutine:

LDA DC00	Inhalt des DATA-Port A in den Akku
ROR	Durch Rechtsrotieren wird Bit 0 in die Carry-Flagge geschoben.
BCC OBEN	Wenn die Carry-Flagge nicht gesetzt ist, war Bit 0 eine Null, also die Joystickfunktion »OBEN«, zu deren Bearbeitung hier verzweigt werden kann.
ROR	Das nächste Rechtsrotieren schiebt Bit 1 in die Carry-Flagge.

```
BCC UNTEN      Abzweigung zur Bearbeitung von »UNTEN«, wenn Bit
               1 nicht gesetzt war.
ROR            Bit 2 ins Carry-Bit
BCC LINKS      und bearbeiten, wenn nicht gesetzt
ROR            Bit 3 in Carry-Flagge
BCC RECHTS     und verzweigen, wenn Bit 3 Null war.
ROR            Zu guter Letzt kommt noch Bit 4 ins Carry-Bit
BCC FEUER      und kann bearbeitet werden, wenn es Null war.
...            weitere Bearbeitung, wenn keine Joystickfunktion
```

Der Vorteil dieser nur 18 Byte langen Unterroutine liegt in ihrer Schnelligkeit: Sie benötigt nur 24 Taktzyklen, wenn nicht verzweigt wird, beziehungsweise 25, wenn verzweigt wird. Natürlich wäre an Stelle von ROR auch die Verwendung von LSR möglich gewesen, denn die herausgeschobenen Bits werden nicht mehr benötigt.

Im Fall, dass man nach einer solchen Abfrage wieder den Ausgangszustand des Akkus oder der Speicherstelle herstellen will, muss man eine entsprechende Anzahl ROR-Anweisungen anschließen, bis Bit 0 wieder in seine Ausgangslage rotiert ist.

Die 16-Bit-Multiplikation

Wir haben in der letzten und in dieser Folge gelernt, wie man 8-Bit-Zahlen miteinander malnehmen kann, um 8- oder 16-Bit-Zahlen zu erhalten. Dabei ist unbefriedigend, dass man sich über jede Zahl Gedanken machen muss, wie man sie am Besten multipliziert. Was fehlt, ist ein allgemein gültiges Programm, das in der Lage ist, jede Zahlenkombination zu verarbeiten (solange es sich um 2-Byte-Integers handelt und das Ergebnis als 16-Bit-Zahl darstellbar ist).

Und da haben wir mal wieder Glück: Gut versteckt befindet sich so etwas bereits fertig in unserem Computer. Ab dez. 45900 ($B34C) liegt im Interpreter solch eine Routine, und ihr Einsprungspunkt ist für uns bei dez. 45911 ($B357). Bevor wir aber detailliert darauf eingehen, soll noch das Prinzip erklärt werden, das dabei genutzt wird.

Jeden Tag rechnen Sie wahrscheinlich völlig automatisch Multiplikationsaufgaben, ohne noch Gedanken daran zu verschwenden, wieviel Schweiß das Erlernen dieser Technik früher mal gekostet hat.

Könnten Sie heute noch jemandem genau erklären, warum man da was wie macht? Genau das müssen wir aber tun, damit der Binärautomat (unser C64) multiplizieren lernt. Nehmen wir mal eine Multiplikation von 16 * 15:

```
 16 * 15
 16

 80
240
```

Dass wir nicht so genau wissen, was wir da tun, liegt am ziemlich komplizierten Zehnersystem. Damit das alles einfacher und überschaubarer wird, wechseln wir mal ins Binärsystem: 16 = 10000, 15 = 1111. Die Aufgabe sieht dann so aus:

```
10000 * 1111
 10000
  10000
   10000
    10000
 11110000
```

Jetzt wird schon deutlicher, was wir getan haben. Der Faktor auf der rechten Seite wurde vom MSB an Bit für Bit durchgesehen. Jedes Mal, wenn wir auf eine 1 gestoßen sind (hier waren nur Einsen), haben wir den links stehenden Faktor notiert. Dabei sind wir von Mal zu Mal um eine Stelle nach rechts gerückt, was mit dem Stellenwert des im rechten Faktor gerade betrachteten Bits zu tun hat. Das geschah so lange, bis alle Bits des rechten Faktors durchgearbeitet waren. Die

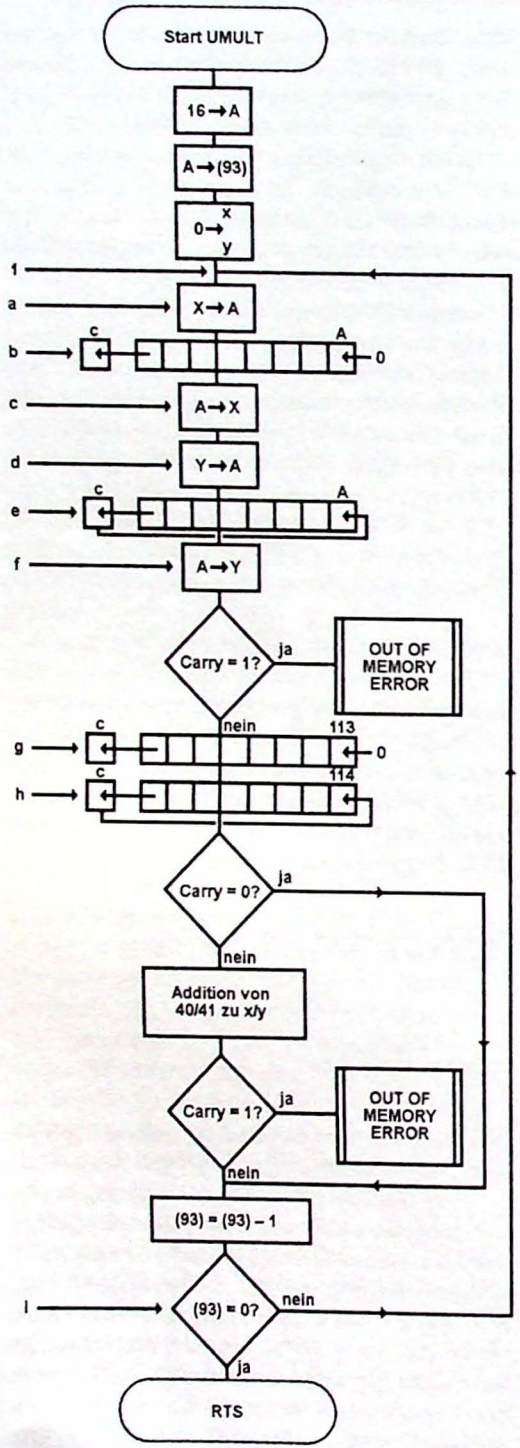


Abb. 21: Flussdiagramm zur Betriebssystemroutine UMULT

Faktor 1: 40/41
Faktor 2: 113/114
Ergebnis: X/Y

sich auf diese Weise ergebende Kolonne wird dann addiert und führt zum Ergebnis. Vergleichen Sie: 240 ist wirklich binär 1111 0000.

Genauso wie hier beschrieben arbeitet das Multiplikationsprogramm. Ein Unterschied tritt auf: es wird nicht bis zum Schluss mit der Addition gewartet, sondern jede neue Zwischenzahl wird sofort addiert. Hier ist die Beschreibung der Interpreterroutine:

Name	**UMULT**
Zweck	Multiplikation zweier 16-Bit-Zahlen
Adresse	$B357 (dez. 45911)
Vorbereitungen	Faktor 1 in $28 / 29, Faktor 2 in $71 / 72
Speicherstellen	$28 / 29, $71 / 72, $5D
Stapelbedarf	keiner
Register	Akku, X- und Y-Register

Diese Routine hier abzudrucken, wäre reine Platzverschwendung. Schalten Sie einfach den SMON ein und verlangen Sie von ihm ein Disassemblerlisting ab B357. Dort haben Sie dann für die weitere Besprechung alles parat. In Abbildung 21 finden Sie das Flussdiagramm der UMULT-Routine.

Das Ergebnis der Multiplikation befindet sich in LSB / MSB-Form in den X / Y-Registern. Programm und Flussdiagramm wollen wir an einem Beispiel nachspielen. Dazu sollen die beiden Zahlen 321 und 65 (binär 0000 0001 0100 0001 und 0100 0001) miteinander multipliziert werden, was bekanntlich 20865 (binär 0101 0001 1000 0001) ergibt. Was Ihnen in Tabelle 23 als undurchdringlicher Bit-Dschungel entgegenstrahlt, ist das schrittweise Verfolgen des Programms im Computerformat, also binär.

In Tabelle 23 sind die Speicheradressen alle dezimal angegeben. Dort finden Sie zunächst die Ausgangslage. In Speicherstelle 40 / 41 steht die ganze Operation über unverändert die Zahl 321. In 113 / 114 finden Sie (wegen des LSB / MSB-Formates umgedreht als 114 / 113) unseren Faktor 65. Akku

	114	113	Y	X	A	93	C	
I	00000000	01000001	00000000	00000000	00010000	00010000	-	Ausgangslage
a				→	00000000		-	
b					00000000		0	
c				00000000	←		0	
d				→	00000000		0	
e					00000000		0	
f			00000000	←			0	
g		10000010					0	
h	00000000						0	
i	00000000	10000010	00000000	00000000	00000000	00001111	0	Ende 1. Schleife
II	00000001	00000100	00000000	00000000	00000000	00001110	1, 0	Ende 2. Schleife
III	00000010	00001000	00000000	00000000	00000000	00001101	0	Ende 3. Schleife
IV	00000100	00010000	00000000	00000000	00000000	00001100	0	Ende 4. Schleife
V	00001000	00100000	00000000	00000000	00000000	00001011	0	Ende 5. Schleife
VI	00010000	01000000	00000000	00000000	00000000	00001010	0	Ende 6. Schleife
VII	00100000	10000000	00000000	00000000	00000000	00001001	0	Ende 7. Schleife
VIII	01000001	00000000	00000000	00000000	00000000	00001000	1, 0	Ende 8. Schleife
IX	10000010	00000000	00000000	00000000	00000000	00000111	0	Ende 9. Schleife
X	00000100	00000000	00000001	01000001	00000001	00000110	1, 0	Ende 10. Schleife
XI	00001000	00000000	00000010	10000010	00000010	00000101	0	Ende 11. Schleife
XII	00010000	00000000	00000101	00000100	00000101	00000100	1, 0	Ende 12. Schleife
XIII	00100000	00000000	00001010	00001000	00001010	00000011	0	Ende 13. Schleife
XIV	01000000	00000000	00010100	00010000	00010100	00000010	0	Ende 14. Schleife
XV	10000000	00000000	00101000	00100000	00101000	00000001	0	Ende 15. Schleife
XVI a		00000000	01010000	01000000	01010000		0	
b	00000000	00000000	01010001	10000001	01010001	00000000	1, 0	Ende 16. Schleife

Tab. 23: UMULT am Beispiel der Multiplikation 321 * 65 = 20865

und Speicherstelle 93 stehen auf 16, dem Bitzähler. In das X- und Y-Register wurde eine Null eingelesen. Im Flussdiagramm ist diese Situation mit einer 1 gekennzeichnet. Ganz unten im Diagramm sehen Sie, dass der Bitzähler 93 erniedrigt und danach geprüft wird, ob er schon gleich Null ist. Daraus folgt, dass die große Schleife 16-mal durchlaufen wird. Den ersten Durchlauf (gekennzeichnet durch kleine Buchstaben) verfolgen wir im Einzelnen.

a) Das X-Register wird zur Bearbeitung in den Akku geschoben.

b) Mittels ASL wird das Bit 7 in die Carry-Flagge geschoben, was einen Carry-Inhalt von 0 bewirkt.

c) Der solchermaßen bearbeitete Akku-Inhalt (der sich hier nicht weiter verändert hat) geht wieder zurück ins X-Register.

d) Nun ist das Y-Register zur Bearbeitung dran. Es gelangt in den Akku.

e) Mittels ROL wandert nun das MSB des X-Registers aus dem Carry-Bit in die 0-Bit-Position des Akkus

f) und alles zusammen wieder ins Y-Register. Insgesamt wird dadurch die 16-Bit-Zahl im X / Y-Register um eine Stellenzahl erhöht, was der Vorbereitung zur Addition dient (Erinnern Sie sich bitte: Die Kolonne der Einzelergebnisse wird ja addiert). Im Diagramm (ohne Buchstabenkennzeichnung) schließt sich hier noch eine Prüfung auf einen eventuellen Überlauf an, der dann mit einer Fehlermeldung beantwortet wird.

g) Nun wird das MSB der Speicherstelle 113 nach links ins Carry geschoben. Das ist auch hier noch eine Null.

h) Anschließend wandert dieser Carry-Inhalt als Bit 0 in Speicherstelle 114. Bit 7 von 114 landet dafür im Carry. Auch hier wird auf diese Weise die ganze 16-Bit-Zahl 113 / 114 um ein Bit nach links geschoben und im nächsten Schritt - im Flussdiagramm wieder ohne Buchstabe - geprüft, ob da eine 1 oder eine 0 ins Carry-Bit geshiftet wurde. Wenn lediglich eine Null auftrat - wie hier -, dann springt das Programm sofort zum Herunterzählen des Bitzählers 93. Tritt aber eine 1 auf, dann addiert sich der Inhalt von 40 / 41 zu X / Y. Hier wird der Zustand der betroffenen Speicherstellen und Register nach dem ersten Schleifendurchlauf gezeigt.

Römisch II bis XVI zeigen nun jeweils den Zustand nach dem 2. bis 16. Durcharbeiten der großen Schleife.

Wenn Sie verstehen möchten, was da passiert, sollten Sie versuchen, Tabelle 23 nur als Kontrolle zu verwenden und ansonsten selbst alle Schritte nachzuvollziehen.

16-Bit-Division

Beim umgekehrten Weg, nämlich der Teilung von zwei 16-Bit-Zahlen, haben wir nicht so viel Glück: Ich konnte keine derartige Routine im Interpreter entdecken. Nun gibt es aber fast in jedem Lehrbuch der Maschinensprache die Vorstellung eines solchen Programms, so dass man sich das Schönste aussuchen kann.

Das Prinzip ist dasselbe, das wir von der normalen Division gewohnt sind: Der Divisor wird Schritt für Schritt vom Dividenden abgezogen. In der Literatur[1] fand ich eine sehr kurze Routine, die ich Ihnen leicht modifiziert als Programm vorstellen will.

In Abbildung 17 wird ein Flussdiagramm dieser Routine gezeigt, und in Tabelle 24 lacht Ihnen wieder das Bit-Gewirr entgegen, das Sie schon von der Multiplikation her kennen, hier aber für die Division gezeigt wird. Damit Sie wissen, wo was hinein- bzw. herauskommt, schauen Sie sich folgende Tabelle an:

A	: B	= C	+ Rest
↑	↑	↑	↑
$57 / 58	$59 / 5A	$57 / 58	$5C / 5D

[1] »Computerspiele und Wissenswertes zum Commodore 64«. Verlag Markt & Technik, Haar bei München 1984. Das ist die Übersetzung des amerikanischen Buches »More on the Sixtyfour« von P. Lücke und jedem Assembler-Programmierer sehr zu empfehlen.

Am folgenden Beispiel soll der Programmverlauf getestet werden: Wir teilen 20867 durch 321. Dabei kommt nach Adam Riese heraus: 65, Rest 2. In folgender Weise wird in die Speicherzellen die Aufgabe eingespeist:

20867	$57	1000	0011	LSB
	$58	0101	0001	MSB
321	$59	0100	0001	LSB
	$5A	0000	0001	MSB
Als Ergebnis findet man dann:				
65	$57	0100	0001	LSB
	$58	0000	0000	MSB
Rest 2	$5C	0000	0010	LSB
	$5D	0000	0000	MSB

Als Bit-Zähler dient hier das Y- Register.
b) Erstes Linksschieben des LSB mittels ASL. Dabei gelangt die 1 in das Carry-Bit.
c) Hineinrotieren der 1 aus dem Carry in das MSB mittels ROL.
d), e) Linksrotieren der 16-Bit-Zahl in $5C / 5D, die jetzt noch 0 ist.
f) Situation am Ende der ersten Schleife. Der Bitzähler ist um 1 reduziert.

Im Folgenden wird dann jeweils die Situation am Ende der Schleife gezeigt. Beim Berechnen der Differenz muss jeweils darauf geachtet werden, dass die Subtraktion einer Zahl als Addition des Zweierkomplements ausgeführt wird. Das haben wir in den Folgen 3 und 4 der Serie kennen gelernt. Allerdings muss an dieser Stelle noch mal gesagt werden, dass die 1, die zum Einerkomplement addiert wird, um das Zweierkomplement zu erhalten, das gesetzte Carry-Bit ist.

Nun dürfte es für Sie eigentlich keine Probleme mehr geben, was das Nachvollziehen der Divisionsroutine betrifft.

Damit dürfen wir getrost die 16-Bit-Arithmetik abschließen. Alle vier Grundrechnungsarten können Sie jetzt programmieren.

Weitere Rechenarten, wie Potenzieren, das Ziehen von Wurzeln, Logarithmen etc. bedingen ohnehin, dass die Argumente oder

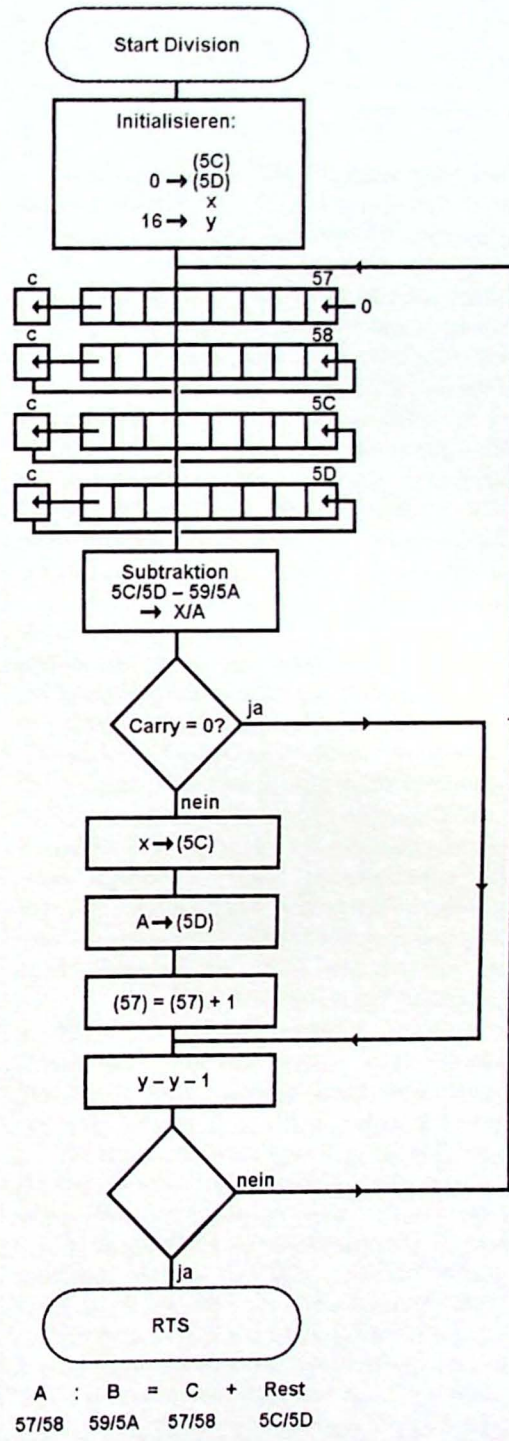


Abb. 22: Flussdiagramm des Programms zur 16-Bit-Division

Ergebnisse keine Integerzahlen sind. Hier werden wir dann mit Fließkommaarithmetik arbeiten und den dazu vorgesehenen Interpreterroutinen.

Das Programmprojekt wird fortgeführt

Im 6. Teil dieser Serie haben wir ein Projekt gestartet, das dort eine Kopfzeile rückholbar unter den oberen ROM-Bereich verschob. Unser Wissen ist seither gestiegen und damit auch unsere Ansprüche. Eine Kopfzeile reicht nicht mehr - jetzt soll es ein ganzer Hilfsbildschirm sein, den wir erst in aller Ruhe erstellen wollen, um ihn dann jederzeit abrufbar unter das Betriebssystem zu packen. Den Aufruf wollen wir wieder mit der USR-Funktion steuern. Diesmal soll aber so programmiert werden, dass der Hilfsbildschirm erhalten bleibt, man ihn also mehrfach einblenden kann.

Über die Nützlichkeit einer solchen Routine braucht man sicherlich nicht viele Worte zu verlieren: Denken Sie da nur mal an Programme, die irgendwelche Tasten mit besonderen Funktionen belegen, für die Sie eine Gedächtnisstütze brauchen, oder ...

Als Programm 3 ist ein kleines Demo-Programm abgedruckt, welches zuerst einen Bildschirm erstellt, dann die Routine »Verschieben« aufruft, den Bildschirm löscht, neu beschreibt und schließlich mit einem weiteren USR den alten Bildschirm einblendet (vorher Programm 4 und 5 laden).

Von nun an können Sie immer - auch im Direktmodus - durch ein USR-Kommando diesen Bildschirm abbilden. Zum Programm aus Folge 6 sind noch zwei Dinge zu bemerken, die hier geändert werden sollen.

Erstens eine Frage: Ist Ihnen der Computer beim Aufruf des Programms mal abgestürzt? Die Wahrscheinlichkeit dafür ist ungefähr 1:60 - wenn nämlich ein Interrupt stattfindet, während die Speicherstelle 1 geändert wird. Obwohl wir erst in den nächsten Folgen auf Interrupts eingehen werden, wollen wir die Wahrscheinlichkeit für so einen Absturz auf Null reduzieren. Eine andere Sache ist der Ort, an dem sich das Programm befand. Es hat sich nämlich herausgestellt, dass anscheinend die Nutzung dieses dort gewählten Speicherbereichs nicht ganz so problemlos ist. Bei einigen Anrufen wurde mir erzählt, dass zumindest der Anfang ab $02A7 bei bestimmten Konstellationen überschrieben wird.

Deswegen packen wir unser Programm ganz unkonventionell nach $6000, von wo Sie es - das beherrschen Sie ja mit dem SMON inzwischen sicher - dorthin schieben können, wo es Ihnen gefällt. Allerdings müssen dann auch die USR-Adressen geändert werden. Aber auch das dürfte für Sie inzwischen kein Problem mehr sein.

Um diese immerhin schon 2000 Byte (je 1000 für den Bildschirm und für das Farb-RAM) zu verschieben, bedienen wir uns der Blockverschiebe-Routine:

Name	**BLTUC**
Zweck	Verschieben von Speicherinhalten im Speicher
Adresse	$A3BF (dez. 41919)
Vorbereitungen	Quelle: Startadresse nach $5F/60, Endadresse +1 nach $5A/5B Ziel: Endadresse +1 nach $58/59
Speicherstellen	$58-5B, $5F, $60, $22
Stapelbedarf	keiner
Register	Akku, X- und Y-Register

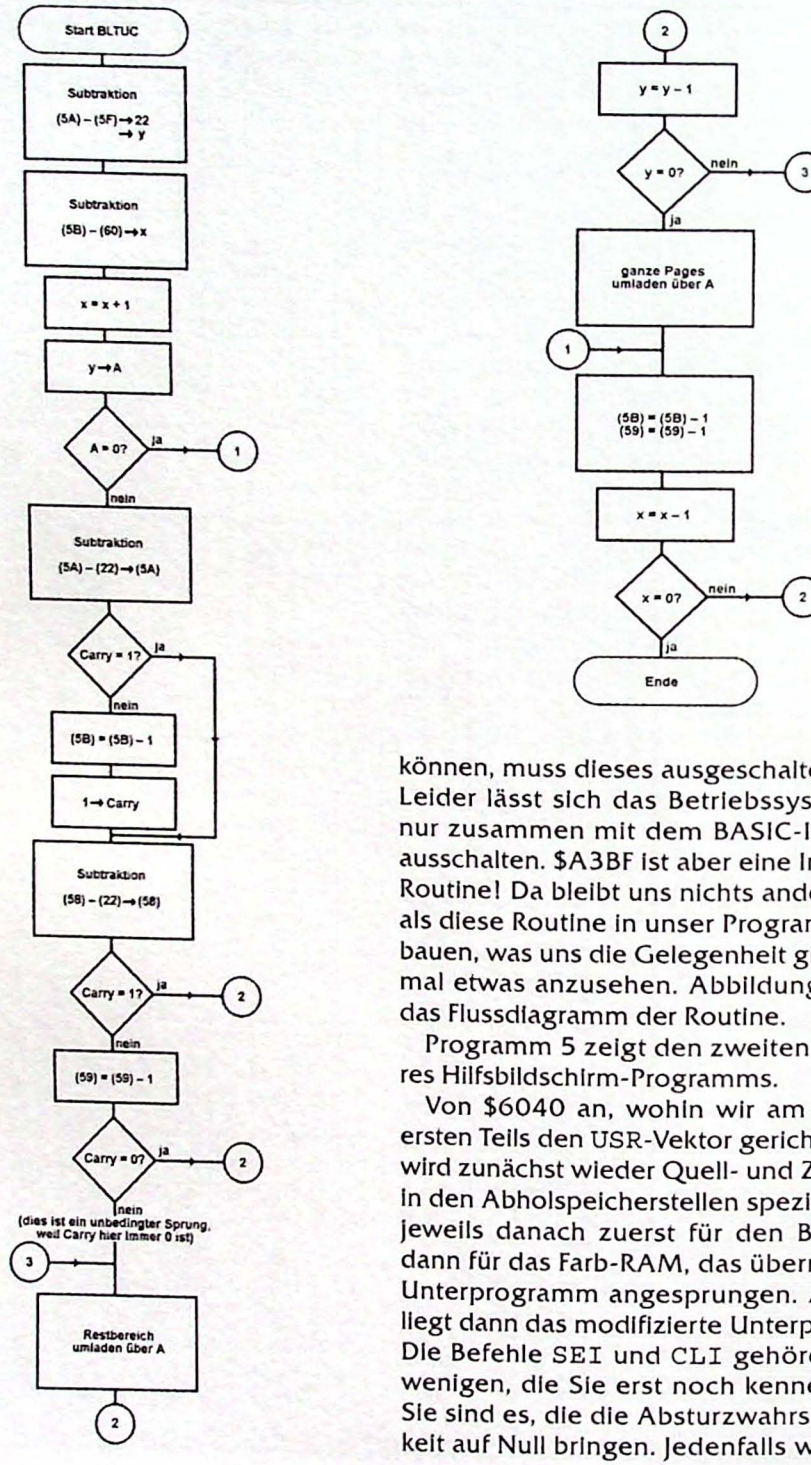
Wieder besteht unser Programm aus zwei Teilen. Im ersten wird der aktuelle Bildschirm nach oben geschoben. Dieser Teil speist lediglich zuerst die Adressen des Bildschirms und des Betriebssystem-ROM in die Abholspeicherstellen der danach aufgerufenen Routine BLTUC und wiederholt diesen Vorgang für die Bildschirmfarbspeicheradressen. Danach verstellen wir noch den USR-Vektor und kehren mit TRS ins BASIC-Programm zurück (siehe Programm 3).

Komplexer ist der zweite Teil. Um nämlich die Informationen unter dem ROM lesen zu

	58	57	5A	59	5D	5C	A	X	Y	C	
Ia	01010001	10000011	00000001	01000001	00000000	00000000	-	00000000	00010000	-	Ausgangsl. n. Init.
b		00000110								1	1. Linksschieben
c	10100011									0	2. Linksschieben
d,e					00000000	00000000				0	3/4. Linksschieben
f	10100011	00000110	00000001	01000001	00000000	00000000	11111110	10111111	00001111	1,0	Ende 1. Schleife
II	01000110	00001100	00000001	01000001	00000000	00000001	11111110	11000000	00001110	1,0,1,0	Ende 2. Schleife
III	10001100	00011000	00000001	01000001	00000000	00000010	11111110	11000001	00001101	1,0	Ende 3. Schleife
IV	00011000	00110000	00000001	01000001	00000000	00000101	11111110	11000100	00001100	1,0,1,0	Ende 4. Schleife
V	00110000	01100000	00000001	01000001	00000000	00001010	11111110	11001001	00001011	1,0	Ende 5. Schleife
VI	01100000	11000000	00000001	01000001	00000000	00010100	11111110	11010011	00001010	1,0	Ende 6. Schleife
VII	11000001	10000000	00000001	01000001	00000000	00101000	11111110	11100111	00001001	1,0,1,0	Ende 7. Schleife
VIII	10000011	00000000	00000001	01000001	00000000	01010001	11111111	00010000	00001000	1,1,0,1,0	Ende 8. Schleife
IX	00000110	00000000	00000001	01000001	00000000	10100011	11111111	01100010	00000111	1,0,1,0	Ende 9. Schleife
Xa	00001100	00000000			00000001	01000110	00000000	00000101		1,0,1,1	
b	00001100	00000001	00000001	01000001	00000000	00000101	00000000	00000101	00000110		Ende 10. Schleife
XI	00011000	00000010	00000001	01000001	00000000	00001010	11111110	11001001	00000101	1,0	Ende 11. Schleife
XII	00110000	00000100	00000001	01000001	00000000	00010100	11111110	11010011	00000100	1,0	Ende 12. Schleife
XIII	01100000	00001000	00000001	01000001	00000000	00101000	11111110	11100111	00000011	1,0	Ende 13. Schleife
XIV	11000000	00010000	00000001	01000001	00000000	01010000	11111111	00001111	00000010	1,0	Ende 14. Schleife
XV	10000000	00100000	00000001	01000001	00000000	10100001	11111111	01100000	00000001	1,0,1,0	Ende 15. Schleife
XVIa	00000000	01000000			00000001	00000011	00000000	00000010		1,1,0,1,1	
b	00000000	01000001	00000001	01000001	00000000	00000010	00000000	00000010	00000000		Ende 16. Schleife = Endlage

Tab. 24: 16-Bit-Division Schritt für Schritt am Beispiel 20867 : 321 = 65 Rest 2

Abb. 23: Flussdiagramm zur Betriebssystemroutine BLTUC



können, muss dieses ausgeschaltet werden. Leider lässt sich das Betriebssystem-ROM nur zusammen mit dem BASIC-Interpreter ausschalten. $A3BF ist aber eine Interpreter-Routine! Da bleibt uns nichts anderes übrig, als diese Routine in unser Programm einzubauen, was uns die Gelegenheit gibt, sie uns mal etwas anzusehen. Abbildung 23 zeigt das Flussdiagramm der Routine.

Programm 5 zeigt den zweiten Teil unseres Hilfsbildschirm-Programms.

Von $6040 an, wohin wir am Ende des ersten Teils den USR-Vektor gerichtet haben, wird zunächst wieder Quell- und Zielbereich in den Abholspeicherstellen spezifiziert und jeweils danach zuerst für den Bildschirm, dann für das Farb-RAM, das übernommene Unterprogramm angesprungen. Ab $6077 liegt dann das modifizierte Unterprogramm. Die Befehle SEI und CLI gehören zu den wenigen, die Sie erst noch kennen lernen. Sie sind es, die die Absturzwahrscheinlichkeit auf Null bringen. Jedenfalls wird zuerst

das ROM aus- und dafür RAM eingeschaltet. Ab $607F bis $60B9 befindet sich die Interpreter-Routine BLTUC.

Darin wird zunächst die Länge des zu verschiebenden Bereichs berechnet, dann festgestellt, ob nur ganze Pages (Seiten) oder auch ein Restbereich verschoben werden soll. Falls ein solcher Restbereich vorhanden ist, wird auch seine Länge berechnet und zuerst dieser verschoben. Daran schließt sich das Verschieben der ganzen Pages an. Das X- und das Y-Register dienen dabei als Zähler.

Ab $60BB schließt sich wieder unsere eigene Routine an, in der wir die ROMs wieder einschalten. Auf diese Weise lassen sich noch mehrere Hilfsbildschirme unter ROM-Bereiche packen. Vielleicht überlegen Sie sich mal dazu einen Weg?

Die ROM-Bereiche als Datenquelle

Die ROM-Bereiche enthalten nicht nur ausgeklügelte Maschinenprogramme, sondern auch eine Menge Daten. Sollten Sie mal in die Verlegenheit kommen, beispielsweise die Zahl PI im MFLPT-Format verwenden zu müssen, dann erfordert das einen ganz schönen Aufwand an Rechen- und Programmarbeit, oder Sie möchten bestimmte Texte wie beispielsweise eine Fehlermeldung verfügbar halten ... und so weiter. Viele von diesen Daten sind schon in der Firmware enthalten, und wir werden im Folgenden festhalten, wo sie sich befinden und welches Format man vorfindet.

Sehen wir uns zunächst Zahlen an (Tabelle 25). Es existieren noch weitere Zahlentabellen in den ROM-Bereichen, die aber selten von Interesse sind. Ebenso wie Zahlen findet man auch Texte als ASCII-Werte im ROM abgelegt (Tabelle 26).

Sollten Sie mal in die Verlegenheit kommen, solche Texte ausgeben zu wollen, dann legen Sie sie nicht noch mal in einer eigenen Texttabelle ab, sondern schöpfen Sie aus dem Fundus, den wir im ROM-Bereich fix und fertig haben.

```
1 REM ********************************** Progr. 3:
2 REM *                                *  Das Demo-
3 REM *          PROGRAMM 3            *  Programm zur
4 REM *                                *  neuen Verschie-
5 REM *   ERSTELLEN UND AUFRUF EINES   *  beroutine. Vor-
6 REM *       HILFSBILDSCHIRMS         *  her müssen Pro-
7 REM *                                *  gramm 4 und
8 REM * HEIMO PONNATH     HAMBURG 1985 *  Programm 5
9 REM **********************************  geladen wer-
10 PRINT CHR$(147) : POKE 785,0 : POKE    den.
786,96 : GOTO 30
15 REM ------- UP-CURSOR SETZEN --------
20 POKE 211,SP : POKE 214,Z : SYS 58640
: RETURN
25 REM -ERSTELLEN DES HILFSBILDSCHIRMS-
30 Z=1 : SP=1 : GOSUB 20 : PRINT "******
**********************************"
40 Z=21 : SP=1 : GOSUB 20 : PRINT "*****
**********************************"
50 Z=10 : SP=7 : GOSUB 20 : PRINT "TEST
FUER DIE VERSCHIEBUNG"
55 REM ---- AUFRUF ZUM VERSCHIEBEN -----
60 A=USR(DUMMY)
65 REM --- BILDSCHIRM NEU BESCHREIBEN---
70 GET A$ : IF A$="" THEN 70
80 PRINT CHR$(147):Z=2:SP=2:GOSUB 20:
PRINT "JETZT SOLLTE DER ALTE BILDSCHIRM"
90 Z=4:SP=2:GOSUB 20:PRINT "UNTER DAS
KERNAL-ROM GESCHOBEN SEIN"
100 PRINT:PRINT:PRINT " -- JEDER{2 SPA-
CE}USR-AUFRUF HOLT DEN --"
110 PRINT " -- HILFSBILDSCHIRM WIEDER.{3
SPACE}"
120 PRINT " -- AUCH IM DIREKT-MODUS{7
SPACE} --"
130 PRINT:PRINT:PRINT"{2 SPACE}PROBIEREN
SIE MAL: A=USR(1) <RETURN>"
140 Z=19 : SP=0 : GOSUB 20 : END
```

Tab. 25: Im ROM stehen nicht nur Programme, sondern auch Tabellen. Hier einige wichtige Zahlen

Startadresse	Format	Zahl
$AEA8	MFLPT	Pi
$B1A5	MFLPT	-32768
$B9BC	MFLPT	1
$B9C1	1-Byte-Integer	3
$B9C2	MFLPT	0,434255942
$B9C7	MFLPT	0,576584541
$B9CC	MFLPT	0,961800759
$B9D1	MFLPT	2,88539007
$B9D6	MFLPT	0,707106781 = SQR(1/2)
$B9DB	MFLPT	1,41421356 = SQR(2)
$B9E0	MFLPT	-0,5
$B9E5	MFLPT	0,693147181 = ln2
$BAF9	MFLPT	10
$BDB3	MFLPT	99999999,9
$BDB8	MFLPT	999999999
$BDBD	MFLPT	1000000000
$BF11	MFLPT	0,5
$BF16	4-Byte-Integer	-100000000
$BF1A	4-Byte-Integer	10000000
$BF1E	4-Byte-Integer	-1000000
$BF22	4-Byte-Integer	100000
$BF26	4-Byte-Integer	-10000
$BF2A	4-Byte-Integer	1000
$BF2E	4-Byte-Integer	-100
$BF32	4-Byte-Integer	10
$BF36	4-Byte-Integer	-1
$BF3A	4-Byte-Integer	-2160000
$BF3E	4-Byte-Integer	216000
$BF42	4-Byte-Integer	-36000
$BF46	4-Byte-Integer	3600
$BF4A	4-Byte-Integer	-600
$BF4E	4-Byte-Integer	60
$BFBF	MFLPT	1,44269504 = 1/ln2
$BFC4	1-Byte-Integer	7
$BFC5	MFLPT	2,14987637E-05
$BFCA	MFLPT	1,43523140E-04
$BFCF	MFLPT	1,34226348E-03
$BFD4	MFLPT	9,61401701E-03
$BFD9	MFLPT	0,0555051269
$BFDE	MFLPT	0,240226385

Startadresse	Format	Zahl
$BFE3	MFLPT	0,693147186 = ln2
$BFE8	MFLPT	1
$E08D	MFLPT	11879546
$E092	MFLPT	3,92767774E-08
$E2E0	MFLPT	1,57079633 = Pi/2
$E2E5	MFLPT	6,28318531 = 2*Pi
$E2EA	MFLPT	0,25
$E2EF	1-Byte-Integer	5
$E2F0	MFLPT	-14,3813907
$E2F5	MFLPT	42,0077971
$E2FA	MFLPT	-76,7041703
$E2FF	MFLPT	81,6052237
$E304	MFLPT	-41,3417021
$E309	MFLPT	6,28318531 = 2*Pi
$E33E	1-Byte-Integer	11
$E33F	MFLPT	-6,8473912E-04
$E344	MFLPT	4,85094216E-03
$E349	MFLPT	-0,0161117018
$E34E	MFLPT	0,034209638
$E353	MFLPT	-0,0542791328
$E358	MFLPT	0,0724571965
$E35D	MFLPT	-0,0898023954
$E362	MFLPT	0,110932413
$E367	MFLPT	-0,142839808
$E36C	MFLPT	0,19999912
$E371	MFLPT	-0,333333316
$E376	MFLPT	1
$E3BA	MFLPT	0,811635157
$E8DA - $E8E9	1-Byte-Integers	Tabelle der Farbkodes
$EB81 - $EBC1	1-Byte-Integers	Tastaturdekodierung: einzelne Tasten
$EBC2 - $EC02	1-Byte-Integers	Tasten mit <SHIFT>
$EC03 - $EC43	1-Byte-Integers	Tasten mit <C=>
$EC78 - $ECB8	1-Byte-Integers	Tasten mit <CTRL>
$ECB9 - $ECE5	1-Byte-Integers	VIC-II-Chip-Registerwerte
$ECF0 - $ED08	1-Byte-Integers	Tabelle der LSBs der Bildschirmanfangsadressen

```
.
,5000        A2 00        LDX #00
,5002        86 5C        STX   5C
,5004        86 5D        STX   5D
,5006        A0 10        LDY #10
,5008        06 57        ASL   57
,500A        26 58        ROL   58
,500C        26 5C        ROL   5C
,500E        26 5D        ROL   5D
,5010        38           SEC
,5011        A5 5C        LDA   5C
,5013        E5 59        SBC   59
,5015        AA           TAX
,5016        A5 5D        LDA   5D
,5018        E5 5A        SBC   5A
,501A        90 06        BCC 5022
,501C        86 5C        STX   5C
,501E        85 5D        STA   5D
,5020        E6 57        INC   57
,5022        88           DEY
,5023        D0 E3        BNE 5008
,5025        60           RTS
-----------------------------------
.?
```

Progr. 4:
Die 16-Bit-Division

Start-adresse	Beschreibung
$A004	CBMBASIC
$A09E - $A19D	Texte der BASIC-Befehlsworte (im letzten Byte ist jeweils Bit 7 gesetzt)
$A19E - $A327	Texte der BASIC-Fehler- und Systemmeldungen (im letzten Byte ist jeweils Bit 7 gesetzt)
$A364 - $A38A	Weitere Systemmeldungen: OK, ERROR, IN, READY, BREAK (das letzte Byte ist jeweils 0)
$ACFC - $AD1D	Fehlermeldungstexte für INPUT: ?EXTRA IGNORED, ?REDO FROM START (das letze Byte ist jeweils 0)
$E460	BASIC BYTES FREE
$E473	**** COMMODORE 64 BASIC V2 ****
$ECE6	64K-RAM-SYSTEM
$F0BD - $F12B	LOAD <RETURN> RUN <RETURN>
$FD10	CBM80

Tab. 26:
Diese Texte sind im ROM als ASCII-Werte abgelegt.

```
.
,6000        A9 00        LDA #00
,6002        85 5F        STA   5F
,6004        A9 04        LDA #04
,6006        85 60        STA   60
,6008        A9 EB        LDA #E8
,600A        85 5A        STA   5A
,600C        85 58        STA   58
,600E        A9 07        LDA #07
,6010        85 5B        STA   5B
,6012        A9 E3        LDA #E3
,6014        85 59        STA   59
,6016        20 BF A3     JSR A3BF
,6019        A9 00        LDA #00
,601B        85 5F        STA   5F
,601D        A9 D8        LDA #D8
,601F        85 60        STA   60
,6021        A9 E8        LDA #E8
,6023        85 5A        STA   5A
,6025        A9 DB        LDA #DB
,6027        85 5B        STA   5B
,6029        A9 D1        LDA #D1
,602B        85 58        STA   58
,602D        A9 E7        LDA #E7
,602F        85 59        STA   59
,6031        20 BF A3     JSR A3BF
,6034        A9 40        LDA #40
,6036        8D 11 03     STA 0311
,6039        A9 60        LDA #60
,603B        8D 12 03     STA 0312
,603E        60           RTS
-----------------------------------
.?
```

Progr. 6:
eine Verschieberoutine

Progr. 5: Zweiter Teil der Verschiebe-routine

```
.
,603F   EA        NOP
,6040   A9 00     LDA #00
,6042   85 5F     STA   5F
,6044   A9 E0     LDA #E0
,6046   85 60     STA   60
,6048   A9 E8     LDA #E8
,604A   85 5A     STA   5A
,604C   85 58     STA   58
,604E   A9 E3     LDA #E3
,6050   85 5B     STA   5B
,6052   A9 07     LDA #07
,6054   85 59     STA   59
,6056   20 77 60  JSR 6077
,6059   A9 E9     LDA #E9
,605B   85 5F     STA   5F
,605D   A9 E3     LDA #E3
,605F   85 60     STA   60
,6061   A9 D1     LDA #D1
,6063   85 5A     STA   5A
,6065   A9 E7     LDA #E7
,6067   85 5B     STA   5B
,6069   A9 E8     LDA #E8
,606B   85 58     STA   58
,606D   A9 DB     LDA #DB
,606F   85 59     STA   59
,6071   20 77 60  JSR 6077
,6074   60        RTS
,6075   EA        NOP
,6076   EA        NOP
,6077   78        SEI
,6078   A5 01     LDA   01
,607A   48        PHA
,607B   A9 35     LDA #35
,607D   85 01     STA   01
,607F   38        SEC
,6080   A5 5A     LDA   5A
,6082   E5 5F     SBC   5F
,6084   85 22     STA   22
,6086   A8        TAY
,6087   A5 5B     LDA   5B
,6089   E5 60     SBC   60
,608B   AA        TAX
,608C   E8        INX
,608D   98        TYA
,608E   F0 23     BEQ 60B3
,6090   A5 5A     LDA   5A
,6092   38        SEC
,6093   E5 22     SBC   22
,6095   85 5A     STA   5A
,6097   B0 03     BCS 609C
,6099   C6 5B     DEC   5B
,609B   38        SEC
,609C   A5 58     LDA   58
,609E   E5 22     SBC   22
,60A0   85 58     STA   58
,60A2   B0 08     BCS 60AC
,60A4   C6 59     DEC   59
,60A6   90 04     BCC 60AC
,60A8   B1 5A     LDA  (5A),Y
,60AA   91 58     STA  (58),Y
,60AC   88        DEY
,60AD   D0 F9     BNE 60A8
,60AF   B1 5A     LDA  (5A),Y
,60B1   91 58     STA  (58),Y
,60B3   C6 5B     DEC   5B
,60B5   C6 59     DEC   59
,60B7   CA        DEX
,60B8   D0 F2     BNE 60AC
,60BA   68        PLA
,60BB   85 01     STA   01
,60BD   58        CLI
,60BE   60        RTS
-----------------------------------
.?
```

Tab. 27: Die in dieser Folge besprochenen Assembler-Befehle

Befehlswort	Adressierung	Byte-Anzahl	Kode		Dauer in Taktzyklen	Beeinflussung von Flaggen
			HEX	DEZ		
LSR	»Akkumulator«	1	1A	26	2	N, Z, C
	absolut	3	4E	78	6	N, Z, C
	0-page-absolut	2	46	70	5	N, Z, C
	abs.-X-indiz.	3	5E	94	7	N, Z, C
	0-page-X-indiz.	2	56	86	6	N, Z, C
ROL	»Akkumulator«	1	2A	42	2	N, Z, C
	absolut	3	2E	46	6	N, Z, C
	0-page-absolut	2	26	38	5	N, Z, C
	abs.-X-indiz.	3	3E	62	7	N, Z, C
	0-page-X-indiz.	2	36	54	6	N, Z, C
ROR	»Akkumulator«	1	6A	106	2	N, Z, C
	absolut	3	6E	110	6	N, Z, C
	0-page-absolut	2	66	102	5	N, Z, C
	abs.-X-indiz.	3	7E	126	7	N, Z, C
	0-page-X-indiz.	2	76	118	6	N, Z, C

Assembler ist keine Alchemie
Folge 10

Erschienen in: 64'er 7/1985
Verantwortlicher Redakteur: Georg Klinge
Autor: Heimo Ponnath

Auf zum Endspurt: Die letzten vier Assembler-Befehle werden in Angriff genommen (dies ist jedoch nicht die letzte Folge des Kurses). Wir steigen ein in ein unerhört interessantes, wenn auch nicht einfaches Thema: die Interrupt-Technik.

Die Assembler-Befehle haben wir bis auf vier noch offen stehende alle behandelt. Diese vier, die alle mit dem Interrupt-Handling zusammenhängen, sollen diesmal unser Thema sein. Um nach längerer Zeit mal wieder auf den Titel dieser Serie zurückzukommen (nämlich die Alchemie!): Wenn wir diese 4 Befehle beherrschen, haben wir den ersten Schritt zum Meister der Assembler-Alchemie getan. Diese vier kleinen 1-Byte-Befehle öffnen uns eine geheime Pforte zu einem Universum an Programmier-Möglichkeiten, von dem wir bisher kaum zu träumen vermochten.

Doch genug der Schwärmerei - erst kommt noch eine Menge Arbeit, die uns wohl mehrere Folgen dieser Serie in Atem halten wird. Zuvor noch eine Bemerkung: Es gibt kaum ein Thema im Rahmen der Programmierung in Assembler, welches so penetrant häufig Abstürze provoziert wie das nunmehr angesteuerte! Falls Sie noch keine RESET-Taste an ihrem Computer haben, wird es nun höchste Zeit. Diese nützlichen Dinger zählen zur Grundausstattung eines Assembler-Alchemisten.

Was sind Interrupts?

Unser Computer ist - solange er eingeschaltet ist - ständig mit irgendwelchen Tätigkeiten beschäftigt. Im Direktmodus hängt er beispielsweise meistens in einer Warteschleife und harrt der Eingaben, im Programm-Modus arbeitet er sich mit Hilfe der Interpreterschleife durch einen BASIC-Befehlstext hindurch und so weiter. Nun werden Sie ja sicher schon festgestellt haben, dass er im Direktmodus auch den Cursor blinken lässt, in beiden Modi die TI$-Uhr weiterzählt und weitere Dinge macht, die anscheinend so nebenher passieren. Schon in der ersten Folge dieser Serie aber haben wir einen Unterschied zwischen Mensch und Computer festgehalten: Der Mensch kann mehrere Dinge gleichzeitig tun, der Mikroprozessor ist nur zu einer Arbeit pro Zeiteinheit fähig. Weil aber diese Zeiteinheiten so unfassbar kurz sind (etwa eine Millionstel Sekunde), haben wir Benutzer den Eindruck der Gleichzeitigkeit.

Wenn dem aber so ist, wie macht es der Computer, dass er beispielsweise ein Programm abarbeitet und trotzdem die TI$-Uhr weiterzählt? Durch Unterbrechungen (to interrupt = unterbrechen) der gerade ausgeübten Tätigkeit. Ein Beispiel aus dem täglichen Leben soll uns das illustrieren:

Sie lesen gerade diesen Artikel, als das Telefon klingelt und ein Freund von Ihnen wissen möchte, was eigentlich Unterbrechungen sind. Während Sie es ihm erklären, fängt in der Küche der Teekessel schrill zu pfeifen an. Sie sagen Ihrem Freund, er möge sich einen Moment gedulden, gehen in die Küche und nehmen den Kessel vom Herd.

Dann kehren Sie ans Telefon zurück und beenden nach einer Weile das Gespräch. Nach dem Auflegen des Telefonhörers setzen Sie die Lektüre des Artikels fort - fest entschlossen, sich nun nicht mehr unterbrechen zu lassen. Kurze Zeit später klingelt jemand an der Tür. Sie lassen sich dadurch nicht stören.

Dieses Gleichnis gibt ziemlich genau wieder, was sich im Computer - nur bei millionenfacher Geschwindigkeit - bei Unterbrechungen abspielt. In Abbildung 24 ist das Schema des Ablaufs grafisch dargestellt.

In gewisser Weise ähnelt das Ganze dem Abarbeiten von Unterprogrammsequenzen. Weshalb programmiert man dann nicht einfach mittels einiger JSR-Aufrufe? Dafür hat L. A. Leventhal einen einleuchtenden Vergleich: »Ein Unterbrechungs-System entspricht etwa einer Telefonklingel. Sie läutet, wenn ein Anruf empfangen wird, so dass man den Hörer nicht laufend abnehmen muss, um festzustellen, ob sich jemand in der Leitung befindet« (L. A. Leventhal, »6502 Programmieren in Assembler«, tewi Verlag, München o. J., S. 121).

Unterbrechungen können dann angefordert und abgearbeitet werden, wenn sie nötig sind - im Gegensatz zu Unterprogrammen, die erst dann berücksichtigt werden, wenn der Programmzähler einen JSR-Befehl erfasst. Um also schnell reagieren zu können, müsste man in einem Programm sehr oft eine Unterroutine anspringen, die auf gewisse Registerinhalte prüft und dann zur Bearbeitung verzweigt oder - bei Nichtvorliegen einer Bedingung - im normalen Programm weiterfährt. Das kostet unnötig Zeit und Speicherraum. Mancher Verkehr des Computers mit Peripherie erfordert so schnelle Reaktionen, dass diese nur durch Unterbrechen des laufenden Programms geleistet werden können.

Ich denke, dass Sie nun die Notwendigkeit von Unterbrechungen erkennen. Fast jede CPU kennt solche Unterbrechungssysteme. Man kann sie durch die Beantwortung folgender Fragen charakterisieren:

1. Welche Unterbrechungs-Eingänge weist die CPU auf?
2. Wie reagiert die CPU auf eine Unterbrechung?
3. Wie bestimmt die CPU die Unterbrechungsquelle, wenn die Anzahl der Quellen größer ist als die Anzahl der Eingänge?
4. Kann die CPU zwischen wichtigen und weniger wichtigen Unterbrechungen unterscheiden?
5. Wie und wann wird das Unterbrechungssystem freigegeben oder gesperrt?

All diese Fragen werden wir im Laufe dieser Serie für unseren Computer ergründen. Einige dieser Charakteristika sind schnell zu zeigen.

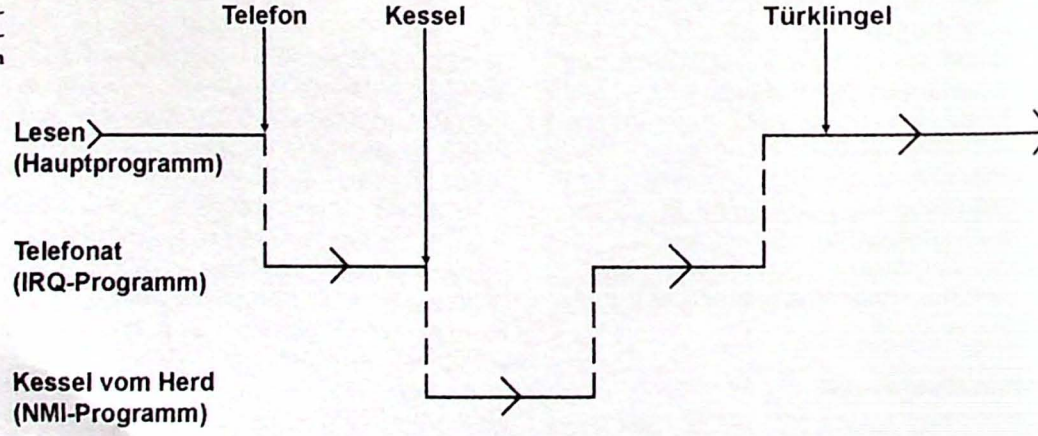


Abb. 24: Unterbrechungen im täglichen Leben

Das Unterbrechungssystem der CPU 6510 / 6502

Zu 1.: Unsere CPU hat genau 2 Eingänge für Unterbrechungen (wenn man RESET außer Acht lässt, was wir im Folgenden meist tun werden).

Zu 3.: Natürlich gibt es weitaus mehr denkbare Unterbrechungsquellen als diese zwei Eingänge, weshalb softwaremäßig eine Registerabfrage (das so genannte Polling) durchgeführt wird, um die Quelle festzustellen.

Zu 4.: Zwischen wichtiger und nicht so wichtiger Unterbrechung kann unsere CPU durch die Priorität der beiden Eingänge unterscheiden.

Wir haben eine so genannte maskierbare Unterbrechung, genannt IRQ, welche per Befehl ignoriert (maskiert) werden kann, und eine andere, nicht maskierbare Unterbrechung, die daher auch NMI (Not Maskable Interrupt = nicht maskierbare Unterbrechung) genannt wird. NMI hat eine höhere Priorität als IRQ und kann deshalb für die wichtigeren Aufgabenstellungen eingesetzt werden.

Zu 5.: Freigegeben oder gesperrt werden kann die IRQ-Unterbrechung durch ein Sperrbit (auch Maskenbit genannt), welches sich als Bit 2 im Flaggen-Register des Prozessors befindet. Das ist die I-Flagge. Für den Empfang der NMI-Unterbrechung kann die CPU nicht gesperrt werden.

Um mal die Parallele zu unserem Beispiel zu ziehen: Das Lesen des Artikels ist die gerade stattfindende Tätigkeit des Computers. Die Telefonklingel signalisiert einen IRQ, der im Folgenden bearbeitet wird.

Das Pfeifen des Teekessels soll einem NMI entsprechen. Wenn dieser dann bearbeitet ist, geht es mit der Abarbeitung des IRQs weiter. Nach Beendigung des Telefonats wird das Unterbrechungs-Sperrbit gesetzt (Sie nehmen sich vor, sich nicht mehr stören zu lassen) und mit der normalen Tätigkeit fortgefahren. Weil der nun folgende IRQ damit maskiert ist, wird das Türklingeln ignoriert.

Die Frage 2, nämlich, wie unsere CPU auf eine Unterbrechung reagiert, blieb noch unbeantwortet. Nun soll sie behandelt werden:

a) Am Ende jedes Befehls überprüft die CPU automatisch den Zustand des Unterbrechungs-Systems. Wenn an einer der beiden Unterbrechungsleitungen eine Anforderung vorliegt und diese auch freigegeben ist, beginnt die Unterbrechung zu wirken.

b) Zunächst wird der Programmzählerinhalt in der Reihenfolge MSB, LSB auf den Stapel geschrieben. Danach wandert noch der Prozessorstatus auf den Stapel (siehe Abbildung 25).

c) Durch Setzen des Unterbrechungs-Sperrbits I werden weitere maskierbare Unterbrechungen (IRQ) unterbunden.

d) Nun holt sich die CPU aus einem Vektor ganz am Ende des Speichers eine Adresse, lädt diese in den Programmzähler und startet auf diese Weise ein Serviceprogramm, das dem auslösenden Anlass Rechnung trägt. In Tabelle 28 sind die zu den Unterbrechungsformen und zum RESET gehörigen Vektoren aufgeführt.

Bevor wir uns weiter mit den so angesteuerten Routinen befassen, wollen wir die 4 Befehle kennen lernen, die uns noch fehlen.

Schlüssel zur Unterbrechungsprogrammierung: CLI, SEI, RTI, BRK

Das Sperren der maskierbaren Unterbrechung IRQ und das Löschen der Maske er-

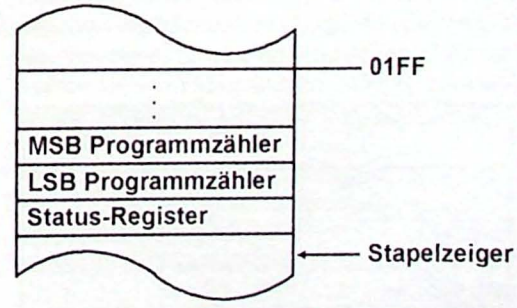


Abb. 25: Die CPU rettet den Programmzähler und das Status-Register beim Eintreten einer Unterbrechung auf den Stapel

folgt durch Setzen oder Löschen des Sperrbits im Prozessorstatus-Register. Dieses Bit, die I-Flagge, kann durch den Befehl CLI gelöscht werden. CLI kommt von »CLear Interrupt mask«, was bedeutet »lösche die Unterbrechungs-Maske«. Immer dann, wenn IRQs zur Bearbeitung durch den Mikroprozessor zugelassen sein sollen, muss damit die I-Flagge gelöscht werden. Wie Sie sehen, ist CLI ein 1-Byte-Befehl mit impliziter Adressierung. Er braucht genau 2 Taktzyklen zur Erledigung seiner Aufgabe.

Wenn wir später eigene Unterbrechungsroutinen schreiben, stehen wir oft vor der Frage, ob wir innerhalb unseres Unterbrechungsprogramms weitere Unterbrechungen zulassen wollen. Manchmal ist das wichtig, beispielsweise bei der Tastaturabfrage. Wie wir vorhin erwähnt haben, sperrt die CPU bei der Annahme von Unterbrechungen automatisch weitere IRQs durch Setzen der I-Flagge. Einer der ersten Befehle der eigenen Unterbrechungsroutine wird dann die Freigabe von Unterbrechungen durch Löschen der I-Flagge sein.

SEI bewirkt das Gegenteil von CLI. Der Befehl setzt die I-Flagge auf 1 (»SEt Interrupt mask«) und verhindert, dass der Mikroprozessor weiteren IRQs seine Aufmerksamkeit schenkt. Das ist in den Fällen wichtig, in denen beispielsweise störungsfrei der Inhalt des Zeichen-ROMs gelesen werden soll oder während der Änderung von Speicherstellen, die die IRQ-Routine benutzt. Wie wichtig das Sperren von IRQs sein kann, haben Sie eventuell bemerkt, wenn Ihnen das Hilfsbildschirmprogramm aus Folge 6 mal abgestürzt ist. Seit der letzten Folge - wo wir die IRQs gesperrt haben - ist Ihnen das sicherlich nicht mehr passiert.

Tab. 28: Unterbrechungsvektoren und ihre Inhalte

Unterbrechungsart	Vektor	Zieladresse
Maskierbare Unterbrechung (IRQ, BRK)	$FFFE / FFFF	65352 $FF48
Reset	$FFFC / FFFD	64738 $FCE2
Nichtmaskierbare Unterbrechung (NMI)	$FFFA / FFFB	65091 $FE43

Ebenso wie CLI ist SEI ein 1-Byte-Befehl mit impliziter Adressierung, und auch er braucht 2 Taktzyklen zur Bearbeitung.

Noch eine Bemerkung zum Verhindern der IRQs: Wir werden später sehen, was alles während der 60-mal pro Sekunde aufgerufenen Unterbrechung erledigt wird. Jede Routine, die SEI verwendet, verbraucht Rechenzeit. Wenn sie so lange dauert, dass eine oder mehrere dieser regelmäßigen IRQs unterbunden werden, kann das unter Umständen zu Störungen von Programmabläufen führen.

In solchen Fällen ist es sinnvoll, in die eigene Routine den Teil der regulären IRQ-Routine einzubauen, der im Programmablauf durch sein Fehlen Störungen verursacht. Meistens kann man dieses Problem aber durch gute Planung eines Programms umgehen.

RTI heißt »ReTurn from Interrupt«, bedeutet zu deutsch also: »kehre aus dem Unterbrechungsprogramm zurück.« Es entspricht in seinem Einsatz etwa dem RTS bei Unterprogrammrücksprüngen. Während RTS aber lediglich den alten Programmzählerinhalt vom Stapel holt (und noch eine 1 addiert), schafft RTI auch noch den alten Inhalt des Status-Registers vom Stapel zurück. Der genaue Ablauf ist wie folgt:

1. alten Prozessorstatus vom Stapel wieder ins Status-Register schieben
2. Stapelzeiger um 1 erhöhen
3. LSB des alten Programmzählers vom Stapel nehmen und zurückschreiben
4. Stapelzeiger um 1 erhöhen
5. MSB des alten Programmzählers vom Stapel nehmen und zurückschreiben
6. Stapelzeiger um 1 erhöhen

Damit ist der Zustand vor der Unterbrechung wieder hergestellt. Auch die I-Flagge ist so automatisch wieder gelöscht, denn vor der Unterbrechung war sie sicher nicht gesetzt gewesen, und der alte Status-Zustand ist ja jetzt wieder vorhanden.

RTI ist ebenfalls ein 1-Byte-Befehl mit impliziter Adressierung. Seine vollständige Bearbeitung dauert 6 Taktzyklen.

Bei eigenen Unterbrechungs-Routinen verwendet man häufig nicht RTI, sondern springt dadurch JMP an eine sinnvolle Stelle des normalen Unterbrechungsprogramms.

Auf diese Weise kann man dann die normalen Arbeitsgänge der vorprogrammierten Unterbrechung oder Teile davon noch ausführen lassen.

Den Befehl BRK (BReaK = Programmunterbrechung) haben wir schon verwendet. Er entspricht in seinem Einsatz etwa dem STOP-Befehl in BASIC und dient wie jeder Befehl dort hauptsächlich dem Testen von Programmen. Tatsächlich unterscheidet sich die Reaktion unserer CPU bei Auftreten eines BRK kaum von der bei einem IRQ. Folgendes passiert:

a) Der Programmzähler wird um 2 erhöht.
b) Bit 4 des Prozessorstatusregisters, die Break-Flagge B, wird auf 1 gesetzt.
c) Das MSB des Programmzählers wird auf den Stapel gebracht und der Stapelzähler um 1 heruntergezählt.
d) Dasselbe geschieht nun mit dem LSB des Programmzählers
e) und mit dem Statusregister.
f) Das Unterbrechungsmaskenbit, die I-Flagge, wird auf 1 gesetzt, um IRQs zu sperren.
g) In den Programmzähler wird nun aus dem Vektor FFFE/FFFF dieselbe Adresse geladen, die auch bei IRQs benutzt wird. Damit startet nun das Programm, das diese Unterbrechung bearbeitet.

Sie sehen, dass der BRK-Befehl ein ziemlich komplizierter Geselle ist. Zwar handelt es sich wieder um einen 1-Byte-Befehl mit impliziter Adressierung, aber er benötigt immerhin 7 Taktzyklen, um all diese Arbeit zu bewältigen.

Wir haben BRK bisher immer zur Programmunterbrechung mit nachfolgender Registeranzeige durch den SMON eingesetzt. Der SMON ist - wie fast jeder Monitor - so programmiert, dass ein BRK zur Registeranzeige führt. Das ist natürlich sinnvoll beim Einsatz von BRK zur Fehlersuche. In dem Moment, wo ein BRK vom Prozessor bearbeitet wurde, kann nur durch die gesetzte B-Flagge von einem IRQ unterschieden werden.

Es ist manchmal nötig, den Unterschied schon zu diesem Zeitpunkt festzustellen. Deshalb verwendet man den nachfolgend beschriebenen Test zu diesem Zweck:

```
PLA          in den Akku wird das zuletzt
             auf den Stapel geschobene
             Prozessorstatus-Register ge-
             holt
PHA          und sogleich wieder zurück-
             geschoben.
AND #$10     durch die AND-Verknüpfung
             mit der Binärzahl 0001 0000
             kann eine eventuell vorhan-
             dene B-Flagge isoliert werden.
BNE BREAK    Falls eine B-Flagge gesetzt war,
             ist der Akku ungleich 0 und die
             Bearbeitung verzweigt zum
             von uns konstruierten BREAK-
             Programm. War der Akku nach
             dieser AND-Verknüpfung
             gleich 0, dann erfolgt keine
             Verzweigung und es handelt
             sich um einen IRQ, zu dessen
             Bearbeitung nun zu springen ist.
```

Es gibt noch eine andere - gebräuchlichere - Möglichkeit, zwischen einem BRK und einem IRQ zu unterscheiden, die allerdings erst zu einem späteren Zeitpunkt des computerinternen Unterprogramms erfolgt. Von dieser zweiten Möglichkeit wird im SMON Gebrauch gemacht, und wir werden sie nachher auch kennen lernen.

Natürlich kann der BRK-Befehl auch zu anderen Zwecken als zur Registeranzeige durch einen Monitor verwendet werden. Es kommt immer darauf an, welches Service-Programm wir dem Computer anbieten. Springt man

aus so einem Service-Programm mittels RTI zurück ins Hauptprogramm, dann muss man berücksichtigen, dass der Programmzähler vor der Sicherung auf dem Stapel um 2 erhöht worden ist. Manchmal sind deshalb noch Korrekturen des Programms nötig.

Ich hoffe, dass Sie diesen Artikel bisher nicht zu frustrierend fanden, denn ständig ist die Rede vom eigenen Unterbrechungs-Programm, und dabei wissen Sie - außer durch BRK - noch gar keine Möglichkeit, einen IRQ oder NMI auszulösen, und Sie sind sicher noch sehr vorsichtig mit dem Gedanken an eigene Unterbrechungs-Routinen. Schließlich ist Ihnen ja noch unbekannt, wie die normale Firmware Unterbrechungen behandelt. Keine Angst: All das werden wir noch klären. Betrachten Sie diese Folge zum Thema Unterbrechungen vielleicht mehr wie ein Handbuch, in dem Sie dann, wenn Ihr Verständnis gestiegen ist, noch mal zurückblättern können:

Wir haben bisher nur betrachtet, wie unsere CPU reagiert, wenn an einem der beiden Unterbrechungs-Eingänge (IRQ und NMI) eine Unterbrechungs-Anforderung vorliegt. Um nun aber selbst ins Geschehen eingreifen zu können, ist es nötig, zu wissen, wie diese Anforderung dorthin gelangt. Das erfordert von uns die Beschäftigung mit anderen Computerbausteinen als der CPU, die bisher im Mittelpunkt unseres Interesses stand.

Woher kommen die Unterbrechungs-Anforderungen?

Quellen für Unterbrechungen können viele genannt werden: Diskettenstation, Datasette, Drucker, Modem, Schaltelemente und so weiter. Um aber eine gewisse Übersicht zu bekommen, sollte man zwischen primären und sekundären Unterbrechungsquellen unterscheiden. Das soll kurz erläutert werden:

Die Diskettenstation beispielsweise ist über den seriellen Port mit dem Computer verbunden. Dieser wiederum steht in direktem Kontakt zu 2 Bausteinen, den CIAs. Erst diese CIAs stehen in direktem Kontakt zur CPU. Alle Unterbrechungs-Quellen, die direkt Signale an die beiden Unterbrechungseingänge unserer CPU senden, sollen künftig »primäre« Quellen genannt werden, die anderen, die nur über solch eine primäre Quelle Unterbrechungs-Anforderungen stellen, werden von uns als »sekundäre« Quellen bezeichnet. Weil wir irgendwo einen Schnitt machen müssen - einmal, um nicht völlig auszuufern in der Erklärung von peripheren Geräten (das soll anderen, kompetenteren überlassen bleiben) und zum anderen, weil ich mich da auch nicht so gut auskenne - werden wir uns im Folgenden auf die primären Unterbrechungsquellen beschränken. Da bleibt aber noch mehr als genug zu tun übrig, und deshalb soll auch nur eine Auswahl dieser Primärquellen detailliert behandelt werden.

Welches sind nun die primären Unterbrechungsquellen? Hier sind sie aufgeführt:

1. Der VIC-II-Chip (MOS 6566/6567 Video Interface Controller)
2. Die beiden CIAs (MOS 6526 Complex Interface Adapter)
3. Die RESTORE-Taste
4. Der Expansion-Port
5. RESET (passt hier nicht ganz hin, woanders aber auch nicht besser)

Den Expansion-Port werden wir nicht behandeln und einen RESET nur ziemlich kurz betrachten, weil es sich dabei eigentlich nicht um eine Unterbrechung im bisher definierten Sinn handelt.

Der VIC-II-Chip als Unterbrechungsquelle

Soweit ich feststellen konnte, kommt der VIC-II-Chip in Bezug auf unsere CPU nur als Anforderer von maskierten Unterbrechungen (IRQs) infrage. Die Handhabung seiner Unterbrechungs-Anforderungen geschieht im VIC-II-Chip durch zwei Register. Vier Ereignisse sind eingeplant, deren Eintreten zur Unterbrechung führen kann:

1. Rasterzeilen-Unterbrechung
2. Kollision eines Sprites mit Hintergrund
3. Kollision von Sprites untereinander
4. Lichtgriffel-Unterbrechung

Die ersten drei Auslöser werden wir uns in kommenden Folgen genau ansehen und dabei vielerlei interessante Möglichkeiten feststellen. Die Option, die der Lichtgriffel bietet, wird nicht behandelt: Meine Kenntnisse auf diesem Sektor sind nur gering (nobody is perfect).

Das so genannte »Interrupt Enable Register« (Unterbrechungs-Zulassungs-Register) des VIC-II-Chips ist Register 26. Es befindet sich in der Speicherstelle 53274 ($D01A) (siehe Abbildung 26). In diesem Register wird festgelegt, ob eines - oder mehrere - der 4 möglichen auslösenden Ereignisse eine Unterbrechungsanforderung an den Mikroprozessor senden soll. Jedem Ereignis ist ein Bit zugeordnet. Ist dieses Bit gleich 1, dann ist die Unterbrechung freigegeben, ist es gleich 0, dann liegt eine Sperrung vor. Die Zuordnung der Bits ist wie folgt:

Bit 0 - Rasterzeilen-IRQ
Bit 1 - Sprite-/Hintergrund-Kollision
Bit 2 - Sprite-/Sprite-Kollision
Bit 3 - Lichtgriffel-IRQ
Bits 4 bis 7 sind ungenutzt und haben immer den Wert 1.

Das Register 25 wird »Interrupt Latch Register« genannt, was etwa zu übersetzen wäre mit »Unterbrechungs-Einrast-Register« (siehe Abbildung 27). Der englische Ausdruck »latch«, der nur umschreibend oder sehr technisch übersetzt werden kann, beschreibt eigentlich recht genau, was in diesem Register geschieht. Ein »latch« ist nämlich so was wie ein Schnappriegel, also ein Riegel, der bei der Betätigung einrastet.

Wenn eines der 4 möglichen Ereignisse eintritt, »schnappt« im dazugehörigen Bit dieses Registers der Inhalt auf 1. Die Bit-Zuordnung ist die gleiche wie in Register 26. Aber das Bit 7 hat hier noch eine Bedeutung: Ist eines der Bits 0 bis 3 auf 1 gesetzt und das dazugehörige Ereignis in Register 26 auch zur Unterbrechung zugelassen (also auch dort gleich 1), dann taucht in Register 25, Bit 7 eine 1 auf. So kann durch einfaches Lesen dieses Bits festgestellt werden, ob ein IRQ durch den VIC-II-Chip ausgelöst wurde.

Will man in diesem Register ein gesetztes Bit löschen, muss man - außergewöhnlich! - eine 1 in die Bitposition schreiben.

Mit Recht erwarten Sie nun eigentlich eine Anwendung des bisher gelernten. Bei Unterbrechungsprogrammen ist es aber dringend nötig, immer den gesamten Komplex im Auge zu haben. Ich habe mich daher entschlossen, zuerst alles zu erklären und dann Anwendungsmöglichkeiten vorzustellen. Ihre Geduld wird auf eine harte Probe gestellt, aber ich hoffe, dass Sie ab der nächsten Folge feststellen, dass sich das Warten gelohnt hat.

7	6	5	4	3	2	1	0
unbenutzt				Lichtgriffel-Unterbrechung	Kollision Sprite / Sprite	Kollision Sprite / Hintergrund	Raster-Unterbrechung

Abb. 26: Das Interrupt-Enable-Register (53274=$D01A) des VIC-II-Chips

7	6	5	4	3	2	1	0
Markierung, ob zugelassene Unterbrechung aufgetreten	unbenutzt			Lichtgriffel-Unterbrechung	Kollision Sprite / Sprite	Kollision Sprite / Hintergrund	Raster-Unterbrechung

Abb. 27: Das Interrupt-Latch-Register (53273=$D019) des VIC-II-Chips

An sich sind die beiden CIAs in unserem Computer völlig identisch. Sie werden aber unterschiedlich eingesetzt. Sehen wir uns zunächst einmal an, was beiden in Bezug auf Unterbrechungen gemeinsam ist, um danach die Unterschiede festzuhalten: Die Unterbrechungs-Steuerung geschieht in Register 13 dieser Bausteine. Dieses Register hat 2 Funktionen: Es bestimmt, ob eine Unterbrechungsanforderung an die CPU gesandt werden soll, und es stellt fest, ob ein Ereignis stattgefunden hat, das zur Unterbrechung führen kann. Die Bedienung dieses Registers ist demzufolge auch etwas unübersichtlich, aber wir haben schon ganz andere Probleme gemeistert.

Die beiden CIA-Bausteine als Unterbrechungsquellen

Sehen wir uns aber zuerst einmal an, welche Ereignisse vom Standpunkt eines CIA-Bausteins als Unterbrechungskriterium dienen können:

1. Unterlauf der Uhr A
2. Unterlauf der Uhr B
3. Die interne Uhr hat eine Alarmzeit erreicht
4. Am SP-Eingang (hängt mit dem seriellen Port zusammen) ist ein bestimmter Zustand erreicht worden
5. An einem Eingang namens FLAG ist ein bestimmter Zustand erreicht

Die Ereignisse 4 und 5 werden wir ebenfalls im Weiteren weitgehend ausklammern.

Nun zum Register 13, dem Unterbrechungs-Kontroll-Register (siehe Abbildung 28). Auch hier gehört zu jedem Ereignis ein Bit. Dabei - um Wiederholungen zu vermeiden - ist die Zuordnung schon durch die eben angegebene Ereignisaufzählung gegeben. Ziehen Sie von der vorangestellten Nummer immer eine 1 ab, und Sie haben die Bitnummer. Die Bits 5 und 6 sind unbenutzt. Bit 7 hat eine dreifache Funktion, die eng mit den anderen Bitinhalten verknüpft ist. Sehen wir uns das mal der Reihe nach an:

Lesen des Registers

Sind Unterbrechungsereignisse aufgetreten, dann sind die dazugehörigen Bits auf 1 gesetzt. Bit 7 ist gleich 1, wenn mindestens ein solches Ereignis stattgefunden hat und außerdem dieses Ereignis als Unterbrechungsauslöser freigegeben ist. Auf diese Weise kann - ähnlich wie beim VIC-II-Chip-Register 25 - festgestellt werden, ob die Unterbrechung durch einen der beiden CIAs angefordert wurde.

Im Unterschied zum VIC-II-Register wird Register 13 aber durch das Lesen gelöscht. Braucht man den Inhalt also noch, sollte man ihn irgendwo zwischenspeichern.

Schreiben in das Register

Bit 7 = 0 erzeugt Sperren. Das erkennt man am besten an einem Beispiel. Nehmen wir an, wir möchten die Unterbrechung sperren, die durch einen Unterlauf von Uhr A erzeugt werden kann. Das betrifft das Bit 0. Wir schreiben in das Register 13 folgende Zahl: `0000 0001`. Wie Sie sehen, ist das Bit 7 gleich 0. Die 1 in Bit 1 bewirkt die Sperrung. Durch die Nullen in den anderen Bits wird bewirkt, dass die anderen Unterbrechungs-Ereignisse nicht beeinflusst werden. Wollten wir alle sperren, dann müssten wir einschreiben: `0001 1111`

Auf diese Weise können selektiv einzelne Unterbrechungen durch Einschreiben der 1 bei gelöschtem Bit 7 gesperrt werden. Bit 7 = 1 erzeugt Freigabe.

Abb. 28: Genereller Aufbau der Unterbrechungs-Kontroll-Register (13) der beiden CIA-Bausteine

7	6	5	4	3	2	1	0
Mehrfunktions-Bit	– unbenutzt –		Flag-Eingang	SP-Eingang	Alarm bei interner Uhr	Unterlauf Uhr B	Unterlauf Uhr A

Auch hier wieder ein Beispiel: Wenn wir ganz gezielt Unterbrechungen durch Unterlauf der Uhr A freigeben wollen, müssen wir die folgende Zahl in Register 13 schreiben: 1000 0001. Bit 7 (= 1) zeigt an, dass diejenigen Unterbrechungen freizugeben sind, deren Bits auf 1 gesetzt sind.
Alle anderen Unterbrechungen, wo also in der dazugehörigen Bitposition der einzuschreibenden Zahl eine 0 steht, bleiben unverändert.

Ein wichtiger Unterschied zwischen den beiden CIAs ist der, dass der Unterbrechungsausgang von CIA 1 mit dem IRQ-Eingang der CPU verbunden ist, wohingegen der entsprechende Ausgang von CIA 2 an den NMI-Eingang unseres Mikroprozessors führt. Daher löst der CIA 1 nur IRQs aus. Deshalb wird er manchmal auch IRQ-CIA genannt. Der andere ist dann der NMI-CIA, weil er nur NMIs anfordern kann.

Der IRQ-CIA

Das Register 13 des IRQ-CIAs (der die Speicherstellen 56320 bis 56335 belegt), liegt in Zelle 56333 ($DC0D). Die einzelnen Bits sind wie folgt zugeordnet:

Bit 0 - Unterlauf Uhr A. Von hier kommt der IRQ, der 60-mal pro Sekunde zur Tastaturabfrage, zum Weiterstellen der TI$-Uhr etc. stattfindet.
Bit 1 - Unterlauf Uhr B. Spielt bei Kassettenoperationen und dem seriellen Port eine Rolle.
Bit 2 - ALARM bei interner Uhr. Spielt beim Zufallszahlengenerator (RND(0)) eine Rolle.
Bit 3 - Hier kommen durch den User-Port Unterbrechungs-Anforderungen.
Bit 4 ist mit dem seriellen Port und der Kassetten-Lese-Leitung verbunden.

Der NMI-CIA

Ebenso kurz und schmerzlos wie beim CIA 1 soll auch das Besondere am CIA 2, dem NMI-CIA (er belegt den Speicher von 56576 bis 56831) vorgestellt werden. Sein Register 13 findet sich in Speicherstelle 56589 ($DD 0D). Die Bits 0 und 1 (Unterläufe der beiden Uhren) spielen beim Senden bzw. Empfangen von Daten über die RS232C-Schnittstelle eine Rolle, Bit 2 (ALARM) wird nicht verwendet. Bit 3 ist direkt mit dem User-Port verbunden, ebenso wie Bit 4. Der NMI-CIA wird uns in seiner normalen Funktion nicht mehr beschäftigen.

Die <RESTORE>-Taste und ein kleines Testprogramm

Die RESTORE-Taste ist direkt mit dem NMI-Eingang unseres Mikroprozessors verbunden. Das ermöglicht es uns, durch einfaches Drücken dieser Taste jederzeit ins Geschehen einzugreifen, ohne uns um Details kümmern zu müssen, beispielsweise ob sich der Computer gerade im Direkt- oder im Programm-Modus befindet und so weiter. Denn NMI hat die höchste Priorität der Unterbrechungen.

Ein kleines Testprogramm soll Ihnen hier noch vorgestellt werden, das Sie vielleicht noch nicht ganz verstehen werden, weil wir erst in der nächsten Folge die eingebauten Serviceprogramme kennen lernen werden. Schalten Sie also den SMON ein und geben Sie das Programm 5 ein (ab $6000).

Am Besten speichern Sie nun das Programm ab und schalten dann mittels des SMON-Kommandos `M 0318` die Anzeige der Bytes ab $0318 ein. Dort steht in den beiden ersten Speicherzellen 47 und FE. Mit dem Cursor fahren Sie in diese Zeile und ändern den Inhalt in 00 und 60, also unsere Programmstartadresse in der LSB/MSB-Form.

Nach einem <RETURN> läuft nun jede NMI-Anforderung über unser Programm. Nun können Sie es ausprobieren, indem Sie mal die <RESTORE>-Taste drücken. Es genügt völlig, alleine diese Taste zu betätigen. Das wirkt - sichtbar durch die Änderung der Rahmenfarbe - in jedem Modus und jederzeit. Eine kleine Merkwürdigkeit ist, dass man manchmal etwas Geduld aufbringen muss, bis man die Wirkung sieht. Ich vermu-

te, dass der NMI so schnell erledigt wird, dass sich mehrere NMIs pro Tastendruck ereignen. Man müsste sich noch eine kleine Routine überlegen, welche die Wirkung etwas verzögert, denn zwei solche EOR-Kommandos nacheinander heben sich gegenseitig auf.

Damit sei's für diesmal genug. In der nächsten Ausgabe werden wir uns die Unterbrechungs-Firmware ansehen und einige Programmbeispiele vorstellen. Zum Schluss noch wie üblich eine Aufstellung mit den besprochenen Befehlen (Tabelle 29).

Ein kleines Testprogramm demonstriert die Wirkung einer Unterbrechung. Durch Drücken der <RESTORE>-Taste wird die Rahmenfarbe geändert.

```
6000 PHA          Mit diesen Befehlen
6001 TXA          retten wir Akku und
6002 PHA          Register auf den Stapel.
6003 TYA
6004 PHA
-------------------------------------------
6005 LDA #$7F     0111 1111 ist das binär.
6007 STA $DD0D    Dadurch werden alle
                  NMIs, die vom CIA 2
                  kommen könnten, ge-
                  sperrt. Erinnern Sie sich:
                  Bit 7 ist Null beim Schrei-
                  ben, also Sperrfunktion
600A LDY $DD0D    Lesen des Registers 13
                  löscht dieses und zeigt
                  uns, ob die NMI-Anfor-
                  derung von dort kam.
600D BMI $601A    Falls NMI-Anforderung
                  vom CIA 2 kam, wird
                  verzweigt,
600F LDA $D020    ansonsten kommt der
                  NMI von der <RESTO-
                  RE>-Taste, und in den
                  Akku wird die Rahmen-
                  farbe eingeladen.
6012 EOR #$0E     Davon ausgehend, dass
                  als Rahmenfarbe 14 vor-
                  liegt, wird diese exklu-
                  siv oder verknüpft zu
                  Null. Ist die Rahmen-
                  farbe 0, dann wird sie
                  wieder 14.
6014 STA $D020    Einschreiben des neu-
                  en Farbwerts
6017 JMP $FEBC    Sprung in den Rest der
                  normalen NMI-Routine
601A JMP $FE72    Sprung in die normale
                  NMI-Routine in dem Fall,
                  dass die Anforderung
                  durch den NMI-CIA kam.
```

Tab. 29: Die Daten zu den letzten Assembler-Befehlen

Befehlswort	Adressierung	Byte-Anzahl	Kode		Dauer in Taktzyklen	Beeinflussung von Flaggen
			HEX	DEZ		
CLI	implizit	1	58	88	2	I-Flagge
SEI	implizit	1	78	120	2	I-Flagge
RTI	implizit	1	40	64	6	alle Flaggen
BRK	implizit	1	00	0	7	B-Flagge vor dem Schieben auf den Stapel, I-Flagge danach

Assembler ist keine Alchemie
Folge 11

Erschienen in: 64'er 8/1985
Verantwortlicher Redakteur: Georg Klinge
Autor: Heimo Ponnath

Noch einmal geht es um Unterbrechungen. Wie geht unser Computer bei den verschiedenen Möglichkeiten vor? Wo kann man selbst einhaken und wie schreibt man eigene Unterbrechungsprogramme? Anhand von zwei Beispielen lernen Sie in dieser Folge, wie man den Ausstieg aus einem Programm verhindert und wie man die Rasterzeilenunterbrechung für eigene Programme einsetzt.

In der letzten Folge haben Sie erfahren, wie unsere CPU mit Unterbrechungen umgeht, welche Sorten von Unterbrechungen es gibt, welches Instrumentarium die Assembler-Sprache zur Behandlung dieser speziellen Technik bietet, und Sie kennen die »primären« Quellen für eine Unterbrechung.

Diesmal wollen wir analysieren, wie die Unterbrechungen softwaremäßig bearbeitet werden, um Wege zu finden, die uns auf möglichst einfache Weise Eingriffe erlauben.

Der normale Verlauf eines IRQ

Neulich hatten wir bereits festgestellt, dass eine IRQ-Anforderung (nach dem Retten des Programmzählers und des Prozessorstatus-Registers sowie dem Setzen der I-Flagge) den Inhalt des Vektors $FFFE/FFFF in den Programmzähler holt.

Dort steht die Adresse $FF48 (dez. 65352) und deshalb startet nun das dort im ROM verankerte Programm, welches wir uns nun im Einzelnen ansehen werden (alle Adressen als Dezimalzahlen; in Abbildung 29 finden Sie das Flussdiagramm dazu).

```
65352 PHA          Zunächst werden der
      TXA          Akku und die Regis-
      PHA          ter X und Y auf den
      TYA          Stapel geschoben.
      PHA
```

Trickreich sind die beiden folgenden Befehle, mit denen das zu Beginn durch die CPU gerettete Statusregister gelesen wird:

```
      TSX          Stapelzeiger ins X-
                   Register
      LDA 260,X    Einladen des Status-
                   Registers
```

Nun wird geprüft, ob die BRK-Flagge gesetzt ist. Wenn das der Fall ist, dann ist der Auslöser ein BRK gewesen, ansonsten ein IRQ:

```
      AND #16      Isolieren der BRK-
                   Flagge
      BEQ 65368    Wenn keine BRK-
                   Flagge, dann über-
                   springen des nächs
                   ten Befehls
65365 JMP (790)    Falls BRK
65368 JMP (788)    Falls IRQ
```

Den vorletzten Sprungbefehl werden wir bei der BRK-Behandlung verfolgen. Interessant für uns ist jetzt der indirekte Sprung bei 65368. Der Vektor 788/789 ($314/315) liegt im RAM! Damit können wir ihn auf eigene Routinen verstellen. Genau hier ist der Ansatzpunkt für nahezu alle Eingriffe in die Unterbrechungsbehandlung. Der voreinge-

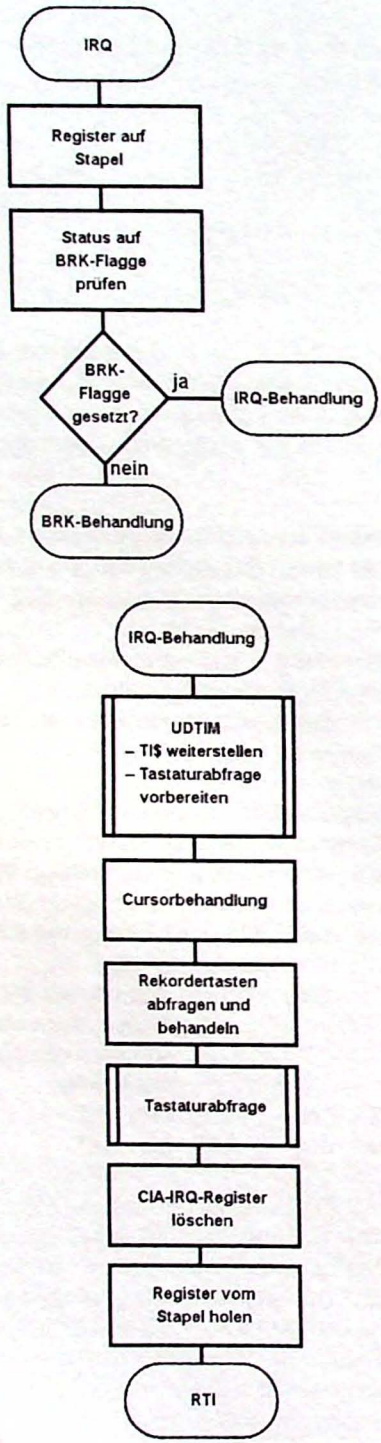


Abb. 29: So bearbeitet der C 64 normale IRQs. Das Programm wird im Text erklärt.

stellte Wert in diesem Vektor ist die Adresse 59953 ($EA31). Das dort angesiedelte Programm wird im Normalfall 60-mal in der Sekunde ausgeführt:

`59953 JSR 65514` Das ist ein Kernal-Sprungbefehl zur Routine UDTIM bei 63131.

In diesem Unterprogramm wird zuerst die Uhr TI$ weitergestellt und dann die Tastaturabfrage vorbereitet.

`59956 bis 60000` In diesem Programmteil erfolgt die Cursorbehandlung.

`60001 bis 60026` Anschließend wird abgefragt, ob eine Rekordertaste gedrückt ist und entsprechende Flaggen bearbeitet.

`60027 JSR 60039` Dieses Unterprogramm dient zur Tastaturabfrage.

Auch in dieser Routine tritt übrigens ein indirekter Sprung nach einem RAM-Vektor auf (655/656 = $28F/290), der normalerweise auf 60232 zeigt, aber auch auf eine eigene Routine verbogen werden könnte. In der Tastaturabfrage ist auch die Überprüfung der <RUN/STOP>-Taste enthalten, die aber nur zusammen mit den im UDTIM-Aufruf voreingestellten Flaggen funktioniert. Deshalb wird das Abschalten der <RUN/STOP>-Taste im Allgemeinen dadurch durchgeführt, dass man den IRQ-Vektor auf 59956 stellt und damit den ersten JSR-Befehl überspringt. Allerdings wird auf diese Weise auch die TI$-Uhr nicht weitergestellt.

`60030 LDA 56333` Das ist das Unterbrechungs-Kontrollregister des IRQ-CIA, das hier durch Auslesen gelöscht wird.

Den Abschluss der IRQ-Routine bildet nun noch das Zurückschreiben der Register:

```
60033 PLA      Zurückholen des
      TAY      Y- und
      PIA
      TAX      des X-Registers
      PLA      sowie des Akkus
60038 RTI      Damit kehrt der Com-
               puter zu dem durch
               den IRQ unterbroche-
               nen Programm zurück.
```

Somit hätten wir's. Nun können wir je nach Bedarf entscheiden, welche von diesen Servicetätigkeiten wir bei einem eigenen IRQ-Programm brauchen: Die Uhr TI$, die Cursorbehandlung, die Tastaturabfrage und die Abfrage der Rekordertasten.

Sehen wir uns nun an, was geschieht, wenn ein BRK-Kommando der Auslöser war.

BRK-Unterbrechung

Abb. 30: Auf diese Weise verläuft ein unvorhergesehener BRK im Sande.

Wir hatten vorhin am Scheideweg zwischen IRQ und BRK den Letzteren links liegen gelassen. Normalerweise verwendet man beim Programmieren in Assembler ja ein Software-Instrument wie zum Beispiel den SMON, der so gebaut ist, dass der BRK-Vektor, welchen wir vorhin kennen gelernt haben ($316/ 317 = 790/791) auf die Registeranzeige weist.

Was geschieht eigentlich, wenn der BRK-Vektor unverändert bleibt, also so, wie er im Einschaltzustand des Computers vorliegt? Dann zeigt er auf Adresse 65126 ($FE66), wo ein Teil der NMI-Routine zu finden ist (Siehe auch Flussdiagramm in Abb. 30):

```
65126 JSR 64789  Sprung ins Programm
                 RESTOR, in dem alle
                 Vektoren (788 - 819)
                 gemäß einer ROM-Lis-
                 te auf ihre Ausgangs-
                 werte gesetzt werden.
      JSR 64931  Sprung in das Pro-
                 gramm I/O-RESET.
```

In diesem Programm werden die beiden CIAs auf die Anfangswerte gestellt.

```
      JSR 58648  Sprung in ein Pro-
                 gramm, welches zu-
                 erst den VIC-II-Chip
                 initialisiert, dann einen
                 Bildschirmeditor-RE-
                 SET durchführt. Nach
                 Beenden dieser Rou-
                 tine ist der Bildschirm
                 gelöscht.
      JMP (40962) Mit diesem indirek-
                 ten Sprung ist die BRK-
                 Unterbrechung been-
                 det. Man sieht aber
                 jetzt schon, dass es
                 sich hier nicht um eine
                 Unterbrechung im ei-
                 gentlichen Sinn han-
                 delt, sondern vielmehr
                 um einen Abbruch. In
                 40962/40963 steht
                 die Adresse des BA-
                 SIC-Warmstarts
                 (58235). Danach be-
                 findet sich der Com-
                 puter im READY-Zu-
                 stand in der Eingabe-
                 Warteschleife.
```

Das Zurückholen der Register und ein RTI erübrigen sich hier, weil ohnehin viele Werte aus dem unterbrochenen Programm inzwischen weitgehend zerstört sind und alle Unterbrechungskontrollregister (VIC-II-Chip und CIAs) neu belegt wurden. Ein unkontrollierter BRK hat also recht fatale Folgen!

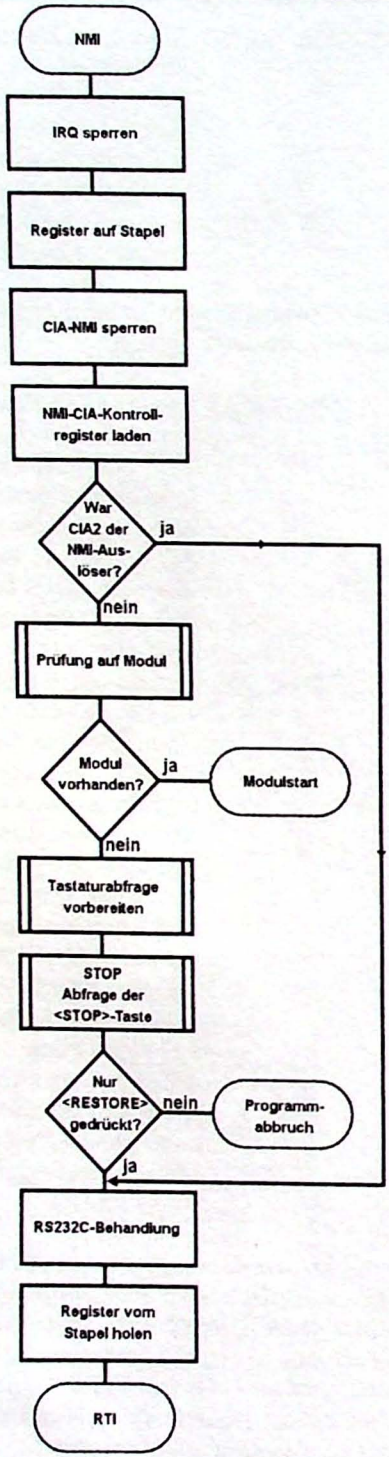


Abb. 31: Flussdiagramm zum Ablauf einer NMI-Unterbrechung

Was macht ein NMI?

Wenden wir uns nun der Firmware zu, die zur Bearbeitung eines NMIs vorgesehen ist (Dazu sehen Sie sich bitte in Abbildung 31 das Flussdiagramm an). In der letzten Folge erfuhren wir, dass auch für diese Unterbrechung am Ende des Speichers ein Vektor vorhanden ist, nämlich $FFFA/FFFB (65530/65531). Dort steht die Adresse 65091 ($FE43), die nun in den Programmzähler gelangt, und damit startet das folgende Programm:

```
65091 SEI          Unterbrechungen
                   niedrigerer Priorität
                   werden gesperrt.
      JMP (792)    Das ist wieder ein für
                   uns sehr interessanter
                   Vektor 792/793
                   ($318/319), der - weil
                   er im RAM-Bereich
                   liegt - verstellbar ist.
                   Genau das haben wir
                   am Ende der letzten
                   Folge mittels des M-
                   Kommandos von
                   SMON festgestellt, um
                   den NMI zu testen,
                   den wir mit der <RES-
                   TORE>-Taste ausgelöst
                   haben. Der voreinge-
                   stellte Wert in diesem
                   Vektor ist die Adresse
                   65095 ($FE47), also
                   direkt der nächste Be-
                   fehl nach dem indirek-
                   ten Sprungbefehl.
65095 PHA          Ebenso wie vorhin
      TXA          beim IRQ werden hier
      PHA          hier die Inhalte des
      TYA          Akkus und der Regis-
      PHA          ter auf den Stapel ge-
                   schoben.
      LDA #127     binär 0111 1111
      STA 56589    Sperrt alle weiteren
                   NMI-Anforderungen
      IDY 56589    NMI-CIA-Kontrollre-
                   gister laden
```

```
BMI 65138
```
Wenn der NMI von der RESTORE-Taste kam, ist Bit 7 des Registers = 0, sonst = 1 (bei NMI-Anforderung durch NMI-CIA).

An dieser Stelle läuft nun das Programm weiter, wenn die <RESTORE>-Taste der NMI-Auslöser war:

```
65110 JSR 64770
```
Das ist ein Unterprogramm, welches prüft, ob ein Modul ab $8000 vorhanden ist.

Dies wird dadurch angezeigt, dass von $8004 bis $8008 die folgenden Werte stehen: 195, 194, 205, 56, 48 (das ist »CBM80«).

```
BNE 65118
JMP (32770)
```
Wenn kein Modulprogramm ab $8000 vorliegt, erfolgt Sprung.
Falls Modul

Wenn ein Modul angezeigt wurde, erfolgt der indirekte Sprung nach dem Vektor $8002/8003, der vom Modul vorgegeben wird. Das kann man auch nutzen, um eigene Maschinenprogramme durch einen Druck auf die <RESTORE>-Taste zu starten. Man muss dann nur in die Speicherstellen $8002 bis 8008 die geforderte Zieladresse beziehungsweise »CBM80« schreiben.

Der nun folgende Abschnitt wird nur angesprungen, wenn die <RESTORE>-Taste der NMI-Auslöser war:

```
65118 JSR 63164
```
Das ist ein Programmteil, der auch schon von der IRQ-Routine (nach dem Weiterstellen von TI$) durchlaufen wird. Hier werden einige Voreinstellungen für die Tastaturabfrage erledigt, die insbesondere die <RUN/STOP>-Taste betreffen.

```
JSR 65505
```
Kernalroutine STOP

Dort befindet sich ein indirekter Sprung über den Vektor 808/809 ($328/329), also auch ein verstellbarer RAM-Vektor. Im Normalfall zeigt dieser Vektor auf 63213 ($F6ED). Dort wird geprüft, ob die <RUN/ STOP>-Taste gedrückt ist.

Eine andere Methode zum Ausschalten von <RUN/STOP> bietet sich hier an, welche die Uhr TI$ ungeschoren lässt:

```
BNE 65138
```
Falls nur die <RESTORE>-Taste (also ohne <RUN/STOP) gedrückt ist, erfolgt ein Sprung.

Waren aber sowohl die <RUN/STOP>- als auch die <RESTORE>-Taste gedrückt, dann folgt nun ein Programmabbruch, der uns schon von BRK her bekannt ist. Hier wie dort endet das Ganze dann mit dem Reset der I/O-Bausteine, des VIC-II-Chips, der Vektoren und des Bildschirmeditors. Das Ergebnis ist ein BASIC-Warmstart.

Ab 65138 befindet sich der Rest der NMI-Routine, auf die das Programm läuft, wenn

1. die NMI-Anforderung nicht von der <RESTORE>-Taste kommt oder
2. zwar von dieser Taste kommt, aber die <RUN/STOP>-Taste nicht gedrückt ist.

```
65138 bis 65211
```
Dieser Abschnitt ist zur Behandlung der RS-232C-Schnittstelle eingerichtet.

```
65212 PLA
      TAY
      PLA
      TAX
      PLA
65217 RTI
```
Abschluss des NMIs durch Rückschreiben des Akkus und der Register vom Stapel

Rückkehr zum unterbrochenen Programm

Wenn Sie sich nun mal unser kleines Demo-Programm aus der letzten Folge ansehen, dann werden Sie feststellen, dass der Programmteil bis $600E lediglich den ersten Teil der normalen NMI-Routine kopiert. Die Prüfung auf das Modul und die <RUN/STOP>-Taste werden übersprungen. Statt dessen erfolgt nach der Abarbeitung des für die <RESTORE>-Taste gebauten Programms das Ende der NMI-Routine ($FEBC = 65212). Im anderen Fall, wenn also die <RESTORE>-Taste nicht der Auslöser des NMIs war, wird in die normale Routine ab 65138 eingemündet.

Eigentlich keine Unterbrechung: RESET

Weil wir alle Unterbrechungen hier bearbeiten wollen, soll auch der RESET angesprochen werden. Es handelt sich dabei aber nicht um eine Unterbrechung im bisher definierten Sinn. Mir fällt allerdings kein Platz in dieser Serie ein, an den der RESET besser passen würde. Ähnlich wie bei NMI und IRQ wird auch hier ein Vektorinhalt in den Programmzähler geladen, der in den höchsten Speicheradressen zu finden ist (auch hierzu wieder ein Flussdiagramm in Abbildung 32).

Dieser Vektor liegt in $FFFC/FFFD. Der Inhalt ist die Adresse 64738 ($FCE2), und genau dort geht das Programm dann weiter:

```
64738 LDX #255    Im ersten Teil wird der
                  Stapelspeicher initia-
                  lisiert.
      SEI         Verhindern von IRQs
      TXS         Stapelzeiger auf $FF
      CLD         Dezimal-Modus aus-
                  schalten (falls er ein-
                  geschaltet war)
      JSR 64770   Das ist wieder das Un-
                  terprogramm, das auf
                  ein Modul prüft.
```

Hier ergibt sich die Möglichkeit, auch beim RESET einzugreifen, indem man die Kennung »CBM80« an die abgefragten Orte packt.

```
      BNE 64751   Falls kein Modul, er-
                  folgt Sprung
64748 JMP(32768)
```

Dieser indirekte Sprung erfolgt nach dem Vektorinhalt von $8000/8001 = 32768/32769. Das ist ein anderer Vektor, als wir ihn vorhin beim NMI hatten (dort war es $8002/8003 = 32770/32771). So kann ein anderer Programmteil angesteuert werden als durch den NMI, was übrigens auch dringend erforderlich ist, weil der Stapelzeiger zerstört wurde.

```
64751               Hier läuft das Pro-
                    gramm weiter, falls
                    keine Modulkennung
                    erkannt wurde.
```

Der ganze Rest dient dem Versetzen des Computers in den Einschaltzustand. Allerdings bin ich davon überzeugt, dass zwischen dem einfachen Aus- und wieder Einschalten des Computers und einem RESET

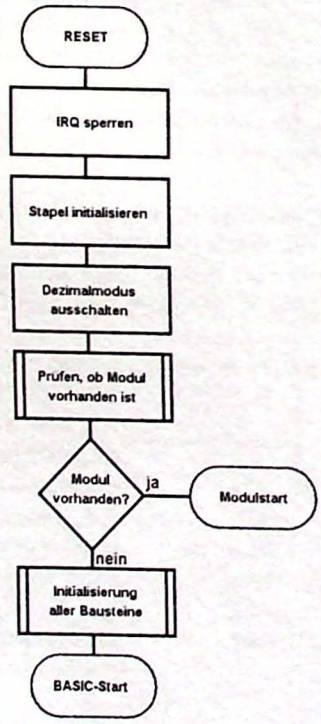


Abb. 32: Diesem Weg folgt ein RESET.

noch irgendein Unterschied bestehen muss. Es hat sich nämlich bei einigen Programmen gezeigt, dass sie nach einem RESET fehlerhafte Verläufe nehmen können, was nach einem totalen Aus- und wieder Einschalten nicht zu beobachten war. Der Grund für diesen Unterschied liegt (für mich) noch im Dunkeln.

In der nächsten Folge werden wir der Sache mit dem Modulstart noch etwas weiter auf den Grund gehen und auch ein interessantes Programm dazu entwickeln.

Assembler ist keine Alchemie
Folge 12

Erschienen in: 64'er 9/1985
Verantwortlicher Redakteur: Georg Klinge
Autor: Heimo Ponnath

Wie macht man Programme Reset-fest? Außerdem schauen wir dem »Rasterzeilen-Interrupt« etwas genauer auf die Finger und entwickeln ein Programm dazu.

Die Sache mit dem Modulstart

Sowohl beim RESET als auch beim NMI haben wir festgestellt, dass der Modulstart-Bereich ab $8000 eine besondere Rolle spielt. In Abbildung 33 finden Sie noch mal zusammengefasst, was sich dort findet, wenn ein Modul vorhanden ist.

Wir wollen im folgenden Beispielprogramm (Programm 6) ein Modul simulieren, indem wir den SMON mittels des RESET anspringen. Der NMI - also die <RUN/STOP>-<RESTORE>-Tastenkombination - soll dabei wirkungslos gemacht werden. Abbildung 34 zeigt ein Flussdiagramm dieses Beispielprogramms.

```
PROGRAMM 6
,6000   A9 2E       LDA #2E
,6002   8D 00 80    STA 8000
,6005   A9 60       LDA #60
,6007   8D 01 80    STA 8001
,600A   A9 41       LDA #41
,600C   8D 02 80    STA 8002
,600F   A9 60       LDA #60
,6011   8D 03 80    STA 8003
,6014   A9 C3       LDA #C3
,6016   A2 C2       LDX #C2
,6018   A0 CD       LDY #CD
,601A   8D 04 80    STA 8004
,601D   8E 05 80    STX 8005
,6020   8C 06 80    STY 8006
,6023   A9 38       LDA #38
,6025   A2 30       LDX #30
,6027   8D 07 80    STA 8007
,602A   8E 08 80    STX 8008
,602D   60          RTS
--------------------------------
,602E   8E 16 D0    STX D016
,6031   20 A3 FD    JSR FDA3
,6034   20 50 FD    JSR FD50
,6037   20 8A FF    JSR FF8A
,603A   20 5B FF    JSR FF5B
,603D   58          CLI
,603E   4C 00 C0    JMP C000
--------------------------------
,6041   68          PLA
,6042   A8          TAY
,6043   68          PLA
,6044   AA          TAX
,6045   68          PLA
,6046   40          RTI
```

Progr. 8: Simulation eines Moduls

Speicherplatz ($)	8000	8001	8002	8003	8004	8005	8006	8007	8008
Inhalt	LSB	MSB	LSB	MSB	C	B	M	8	0
	RESET-Vektor		NMI-Vektor						

Abb. 33: Diesen Inhalt müssen die Speicherstellen $8000 bis $8008 haben, damit ein Modulstart stattfindet.

Achten Sie bitte darauf, dass Sie nach dem Eintippen des Programms abspeichern und - natürlich - dass die SMON-Version ab $C000 im Speicher vorliegt. Mit SYS 24576 starten Sie unser Programm, in dem durch diesen SYS-Befehl zunächst nach $8000/1 die Startadresse einer neuen RESET-Serviceroutine geschrieben wird und nach $8002/3 die der neuen NMI-Routine. Außerdem wird die Modulkennung in die vorgeschriebenen Speicherplätze eingetragen. Wenn Sie nun mal die <RESTORE>-Taste - oder <RUN/STOP> und <RESTORE> - drücken, passiert offensichtlich nichts. Das liegt daran, dass unser Programm lediglich die auf den Stapel gelegten Register wieder zurückholt und aus der Unterbrechung mit RTI ins normale Geschehen zurückkehrt.

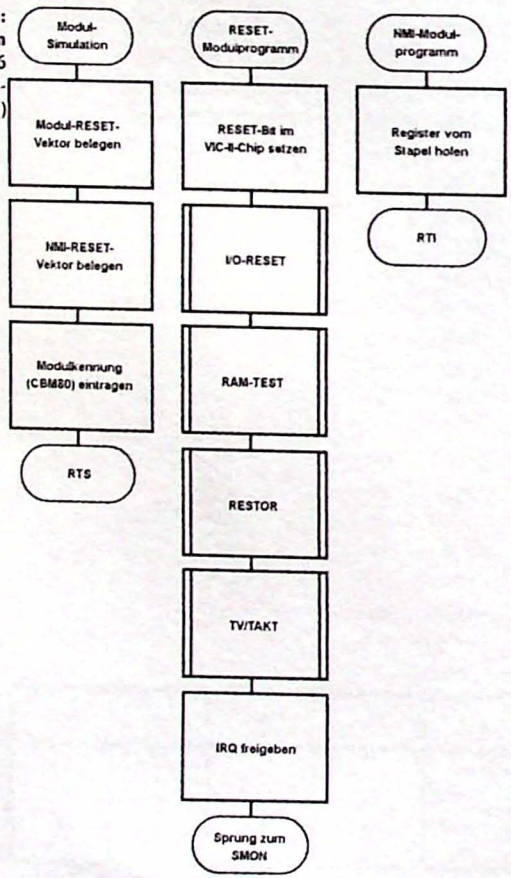


Abb. 34: Flussdiagramm zu Programm 6 (Modulsimulation)

Haben Sie einen RESET-Taster eingebaut? Dann drücken Sie doch mal drauf. Zunächst erkennen Sie den normalen RESET-Verlauf. Dann meldet sich aber nicht wie gewohnt die Nachricht »CBM-BASIC ...«, sondern der SMON mit einer Registeranzeige. Das RESET-Programm ab $602E folgt dem Firmware-Programm. Lediglich der letzte Sprungbefehl ist anders und führt statt ins BASIC in den SMON. Der SMON wird fehlerfrei funktionieren, solange Sie nicht versuchen, mit dem X-Kommando wieder ins BASIC zurückzukehren. Dann wird Unsinn passieren, denn auf einen Start mittels RESET ist der SMON nicht gefasst, und in den Speicherstellen, die sonst eine Rückkehradresse enthalten, befindet sich nichts Sinnvolles. Es ist daher auch nicht möglich, den SMON wieder zu verlassen - außer durch Speicherstellenmanipulationen oder die Notbremse: Aus- und wieder Einschalten.

Auf diese Weise (und mittels eines Autostarts) sichern sich Softwarehäuser manchmal gegen unbefugtes Kopieren ihrer Programme.

Nutzung der Unterbrechungen

Sowohl was die Hardware als auch die Firmware für die Unterbrechungsbehandlung angeht, haben wir nun einen guten Überblick gewonnen. Es ist jetzt an der Zeit, dass wir uns ansehen, auf welche Weise man dieses Reservoir an vielfältigen Möglichkeiten für sich nutzen kann. Dazu soll uns ein Überblick dienen:

1. Auslösung der Unterbrechung durch Hardware-Einwirkungen:

Da hätten wir beispielsweise den User-Port oder den Expansion-Port, über die wir per CIA Unterbrechungen anfordern können. Um es gleich zu sagen: Damit werden wir uns nicht auseinandersetzen. Meine Kenntnisse auf diesem Gebiet sind zu dünn.

2. Unterbrechungsauslösung per Software: Damit haben wir immer noch ein weites Feld von Möglichkeiten vor uns:

2.1 Vorgesehene Nutzungen des IRQ
- mittels des VIC-II-Chips: Da können wir uns auf den Rasterzeileninterrupt, die Sprite-/Hintergrund- oder die Sprite-/Sprite-Kollision stützen.
- mithilfe des CIA 1: Da ist es vor allem der 60-mal pro Sekunde auftretende Timer-A-Unterlauf, der uns interessieren soll.

2.2 Vorgesehene Nutzungen des NMI
- CIA 2: Lässt man die RS232C-Schnittstellenbehandlung außer Acht, dann gibt es keine vorgesehene Nutzung.
- RESTORE: Zusammen mit der Taste <RUN/STOP> kann man die vorgegebene Routine verändern, wie wir es schon in einigen Beispielen gezeigt haben.

Wir können außerdem noch unterscheiden zwischen Nutzungen, die periodisch stattfinden sollen (zum Beispiel eine spezielle Tastaturabfrage) und solchen, die stochastisch (= zufallsabhängig) oder willkürlich erfolgen (zum Beispiel Drücken der <RESTORE>-Taste). Beides ist auch durchführbar bei:

2.3 Nicht vorgesehene Nutzung der Unterbrechungen

Da bietet sich vor allem der meistens völlig brach liegende CIA 2 mit seinen beiden Timern und der Alarmfunktion an. Wenn Sie aber erst einmal vertraut mit der Unterbrechungs-Programmierung sind und auch etwas Zeit zum Tüfteln investieren, finden Sie bestimmt noch eine ganze Menge weiterer Möglichkeiten. Bei mehreren gleichartigen Unterbrechungsanforderungen (zum Beispiel IRQs) muss noch ein Weg gefunden werden, wie zwischen den dann vielleicht anfallenden unterschiedlichen Service-Routinen differenziert werden kann. Denkbar wären beispielsweise Aufgabenstellungen wie:
- Jeder 3. Timer-IRQ soll den Joystick abfragen
- RESTORE + h soll den Hilfsbildschirm zeigen
- RESTORE + z soll den aktuellen Bildschirm wieder restaurieren
- etc.

Sie sehen: Eine große Menge Arbeit wartet auf uns. Nicht zu allen Möglichkeiten werde ich hier Beispielprogramme zeigen.

Außerdem dürfen die dann auch nicht zu undurchsichtig sein, und man sollte möglichst den Erfolg eines solchen Demo-Programms auf dem Bildschirm erkennen können.

Trotzdem hoffe ich, dass die nachfolgend und in der nächsten Folge gezeigten Programmlösungen ausreichen, um Ihnen die Unterbrechungs-Behandlung mit eigenen Routinen durchschaubar zu machen. Ich will Ihnen aber nicht verschweigen, dass auch mir noch längst nicht alle Geheimnisse der Unterbrechungsprogrammierung offenbar geworden sind. Oft finde ich mich unversehens in Programm-Sackgassen wieder. Das soll Ihnen als kleiner Trost dienen, wenn Sie mal nach dem 1001. Absturz müde und mit rauchendem Kopf vor Ihrem Commodore-Ungeheuer sitzen.

Ein Programm zum VIC-II-IRQ

Sehr schöne Effekte lassen sich durch eine periodische IRQ-Anforderung per Rasterzeileninterrupt mittels des VIC-II-Chips erzielen. Deshalb ist so was auch ein beliebtes Objekt für Demos von Unterbrechungsprogrammen. Als Ziel setzen wir uns, einen Bildschirm zu konstruieren, dessen Rahmen in allen Farben schillert.

Der VIC-II-Chip besitzt diese Möglichkeit: Man kann dem Kathodenstrahl, der über den Monitor huscht, um das Bild zu erzeugen, über zwei Register folgen - die Rasterregister, wo jede Rasterzeile mitgezählt wird. Ohne an dieser Stelle allzusehr in die Einzelheiten einzugehen, soll hier nur bemerkt werden, dass die Nummerierung dabei etwa von 0 bis 280 geht, weil auch der Rahmen und nicht sichtbare Teile des Bildschirms vom Strahl überstrichen werden.

Wo das Textfeld anfängt, ist von Monitor zu Monitor (oder Fernseher) etwas unterschiedlich. Bei mir beginnt es oben in Rasterzeile 50 und endet unten bei Zeile 248.

Sollten die im Beispielprogramm 7 nachher voreingestellten Randwerte bei Ihnen also anders sein, können Sie sie durch einige später noch angegebene POKEs ändern. Die beiden Rasterzeilenregister sind:

$D012 (53 266) und $D011 (53 265)

Von $D011 ist allerdings nur das Bit 7 als MSB der Rasterzeilenzahl für uns von Bedeutung. Abbildung 35 soll diese Belegung deutlich machen.

Das Interessante an diesen Registern ist nun, dass man auch in sie schreiben kann. Die auf diese Weise festgelegte Rasterzeile ist dann der Auslöser des IRQs, falls dieser im Interrupt-Enable-Register $D01A freigegeben wurde (das kennen wir noch aus der letzten Folge).

Damit kann also unsere primäre Unterbrechungsquelle (der VIC-II-Chip) programmiert werden. Halten wir die zwei Schritte dazu noch mal fest:

1. Rasterzeile, bei der ein IRQ ausgelöst werden soll, durch Einschreiben in die Register $D012 und Bit 7 von $D011 festlegen
2. Freigeben des Rasterzeileninterrupts durch Einschreiben von 1000 0001 in das Interrupt-Enable-Register $D01A

Der nächste Schritt betrifft die Bearbeitung des IRQs durch die CPU. Wie wir vorhin gesehen haben, springt das Programm beim IRQ mittels eines indirekten Sprungs, der auf den Vektor 788/9 ($314/5) zugreift. Dieser Vektor muss nun auf die eigene Routine verbogen werden, also:

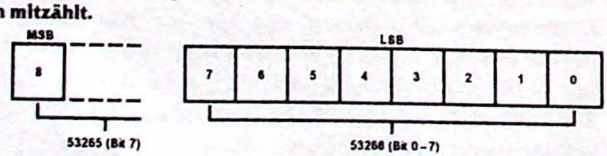


Abb. 35: So sieht das 9-Bit-Register im VIC-II-Chip aus, das die Rasterzeilen mitzählt.

3. Vektor $314/5 auf die IRQ-Service-Routine richten

Damit wären alle Vorbereitungen getroffen. Der Rest liegt nun ganz bei uns - beziehungsweise bei dem von uns zu schreibenden Service-Programm.

Als Abbildung 36 finden Sie das Flussdiagramm unseres Beispielprogramms, das im Folgenden entwickelt wird und das Sie als Programm 7 am Ende des Artikels im Überblick abgedruckt finden.

Gehen wir nun an die Realisierung. Zunächst die Initialisierung, die wir bei $6000 (also durch SYS 24576 zu starten) beginnen lassen:

```
6000 SEI          Sperren von IRQs

Schritt 3:
6001 LDA #$28     LSB der IRQ-Routine
6003 STA 0314     in IRQ-Vektor-LSB
6006 LDA #$60     MSB der IRQ-Routine
6008 STA 0315     in IRQ-Vektor-MSB

Schritt 1:
600B LDA #$F8     Rasterzeile, bei der das
                  Textfenster endet. Von
                  da an soll der Rahmen
                  schwarz sein.
600D STA D012     in Rasterzeilen-Register
                  (LSB) schreiben
6010 LDA D011     Register mit dem MSB
                  des Rasterzeilenzählers
6013 AND #$7F     0111 1111 löscht Bit 7
6015 STA D011     Zurückschreiben. Damit
                  ist die Rasterzeile, die
                  den IRQ auslösen soll,
                  festgelegt.

Schritt 2:
6018 LDA #$81     1000 0001 wird nun ins
                  IRQ-Enable-Register
                  geschrieben, um den
                  Rasterzeilen-IRQ
                  zuzulassen.
```

Festlegen einiger Startwerte:

```
601D LDA #$00    Farbe Schwarz
601F STA D020    in Rahmen schreiben
6022 LDA #$04    Streifenbreite in Merk-
6024 STA 02      register schreiben
6026 CLI         IRQ freigeben
6027 RTS         Ende der Initialisierung
```

Von nun an laufen alle IRQs über unsere eigene Routine, die bei $6028 beginnt. Zuerst müssen wir prüfen, ob die Unterbrechung vom VIC-II-Chip kommt oder vom CIA 1:

```
6028 LDA D019    IRQ-Request-Register
                 des VIC-II-Chips (sie-
                 he letzte Folge). Dort
                 ist Bit 7 gesetzt, wenn
                 die Anforderung vom
                 VIC-II-Chip kam.
602B STA D019    Zurückschreiben
602E BMI 6037    Sprung, falls VIC-IRQ,
                 sonst CIA-IRQ
```

Bearbeiten eines CIA-IRQs

```
6030 LDA DC0D    Löschen des CIA-1-
                 Unterbrechungs-Kon-
                 trollregisters
6033 CLI         IRQ zulassen. Damit
                 können innerhalb ei-
                 nes CIA-IRQs auch un-
                 sere VIC-IRQs gesche-
                 hen.
6034 JMP EA31    Bearbeitung des CIA-
                 IRQs durch die nor-
                 male Routine
```

Unser Programm für VIC-II-IRQs:

```
6037 LDA D012    Rasterzeilen-Register
                 laden, um festzustel-
                 len, welche Zeile den
                 IRQ auslöste
603A CMP #$F8    Vergleich mit Ende des
                 Textfensters
603C BCS 604F    Wenn unterhalb des
                 Textfensters: Sprung
```

Der folgende Programmteil ist wirksam, wenn der IRQ-Auslöser eine Zeile in Höhe des Textfensters war:

```
603E CLC         Addition vorbereiten
603F ADC 02      Streifenbreite aus dem
                 Merkregister addieren
6041 STA D012    Neuen Wert in Raster-
                 zeilen-Register schrei-
                 ben
```

Damit wird eine neue Rasterzeile als IRQ-Auslöser festgelegt, welche um eine Streifenbreite tiefer liegt als die vorhergegangene. Es folgt eine kleine Verzögerungsschleife, die aber nur zum Experimentieren eingebaut wurde:

```
6044 LDY #$03    Schleifen-Startwert
6046 DEY         Herunterzählen
6047 BNE 6046    NEXT Y bis Y = 0
```

Ändern der Rahmenfarbe bis zum nächsten Raster-IRQ

```
6049 INC D020    Farbkode + 1. Wenn Ko-
                 de im Rahmenfarbre-
                 gister > 15 wird, fängt
                 wieder Farbkode 0 an,
                 weil die Bits 5-7 kei-
                 ne Funktion haben.
```

Rücksprung in den Rest der normalen IRQ-Routine:

```
604C JMP EA81    Siehe unsere Unter-
                 suchung der IRQ-Firm-
                 ware.
```

Damit ist der Rahmen in Höhe des Textfensters behandelt. Es schließt sich nun der Teil an, der die Rahmenbereiche unter- und oberhalb bearbeitet:

```
604F LDA #$00    Farbkode Schwarz in
6051 STA D020    Rahmenfarb-Register
6054 LDA #$32    Rasterzeile, bei der das
                 Textfenster oben be-
                 ginnt
6056 STA D012    In Rasterzeilen-Regis-
                 ter schreiben
```

```
6059 JMP EA81    Abschluss durch Sprung
                 zum Ende der norma-
                 len IRQ- Routine
```

Damit ist festgelegt, dass die Rahmenfarbe ober- und unterhalb des Textfensters schwarz wird.

Unsere eigene Routine ist jetzt abgeschlossen. Zum guten Ton gehört es, dem Benutzer auch die Möglichkeit zu bieten, diese Routine wieder abzuschalten. Das erfolgt im letzten Programmteil, der mittels SYS 24688 aktiviert werden kann:

```
605C SEI          IRQ sperren
605D LDA #$00     Raster-IRQ
605F STA D01A     abschalten
6062 LDA #$31     IRQ-Vektor
6064 STA 0314     restaurieren
6067 LDA #$EA
6069 STA 0315
606C LDA #$0E
606E STA D020
6071 CLI
6072 RTS
```

Unser Programm ist komplett. Speichern Sie es bitte vor dem Starten ab. Nach dem SYS 24676 finden Sie einen hübschen bunten Rahmen vor, der oberhalb und unterhalb des Textfensters schwarz ist. Besonders gut - finde ich - sieht das Ganze aus, wenn man die Hintergrundfarbe des Textfensters auch auf Schwarz setzt. Das Programm erlaubt noch einige Experimente: Durch POKE-Kommandos in die Speicherstelle 2 kann die aktuelle Streifenbreite variiert werden, durch POKEs in die Zelle 24645 der Startwert der Verzögerungsschleife.

Probieren Sie's doch mal aus. Eine Erkenntnis werden Sie gewinnen: In der Unterbrechungs-Programmierung spielt die Zeit eine wichtige Rolle. Das zeigt sich auch, wenn man zum Beispiel Cursorbewegungen durchführt: Die Streifen fangen an zu wandern.

Weitere Möglichkeiten zum Experimentieren sind gegeben, wenn Sie die Rasterzeilen verändern, die den oberen und unteren Rand des Textfensters markieren:

Durch POKE 24661,Zahl verschieben Sie die obere, durch POKE 24635,X : POKE 24588,X die untere Rasterzeile, von der an alles schwarz ist. Wie schon vorhin erwähnt, habe ich im Programm diese Werte auf 50 beziehungsweise 248 fi-

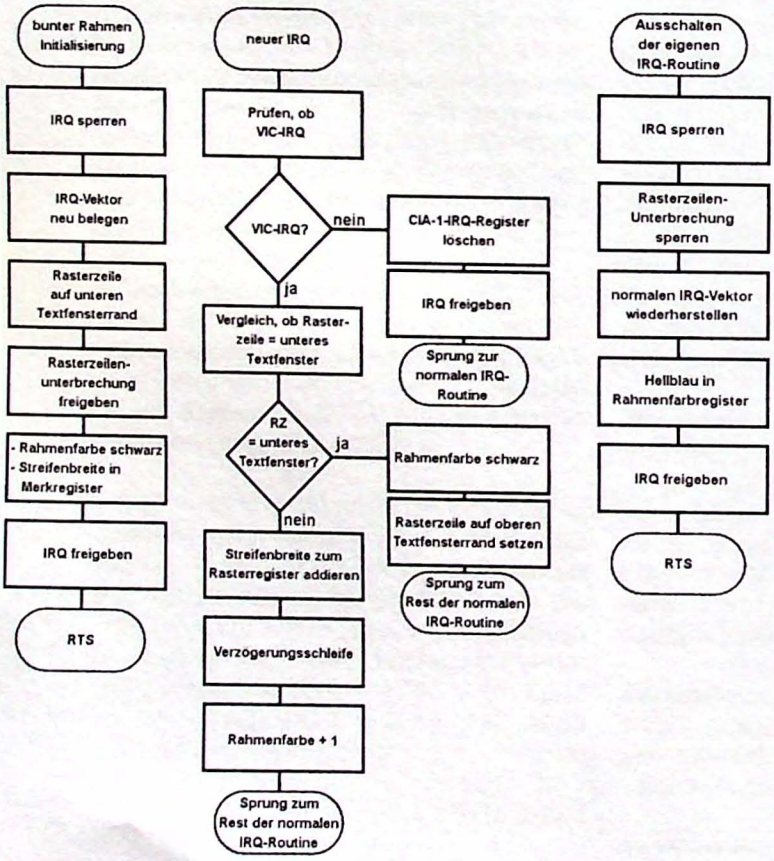


Abb. 36: Flussdiagramm zum im Text vorgestellten Programm 7. Ein bunter Rahmen wird erzeugt.

xiert, weil genau dort auf meinem Monitor das Textfenster liegt.

Mit diesem Beispiel sollte es Ihnen nun möglich sein, auch andere Unterbrechungsprogramme zu schreiben, die sich der Rasterzeilen-Unterbrechung per VIC-II-Chip bedienen.

Eine Bemerkung sollte ich Ihnen noch auf den Weg Ihrer eigenen Versuche mitgeben: Der Elektronenstrahl, der über den Bildschirm saust und beim Erreichen des von uns bestimmten Rasterzeilenwertes zum Auslösen des IRQs führt, ist enorm schnell. Die Serviceprogramme dürfen deshalb nicht zu lang sein, sonst steht der nächste IRQ schon wieder an, bevor der Vorangegangene bearbeitet ist.

In der kommenden Folge sollen Beispiele für andere Unterbrechungsformen vorgestellt werden, die durch die CIAs und die <RESTORE>-Taste angesprochen werden.

Danach soll es um die Anwendung des bisher Gelernten gehen, wobei wir uns wieder mehr den Interpreter-Routinen und auch den Kernal-Möglichkeiten zuwenden wollen.

Progr. 9: Das im Artikel entwickelte Programm auf einen Blick

```
PROGRAMM 9
,6000   78          SEI
,6001   A9 28       LDA #28
,6003   8D 14 03    STA 0314
,6006   A9 60       LDA #60
,6008   8D 15 03    STA 0315
,600B   A9 F8       LDA #F8
,600D   8D 12 D0    STA D012
,6010   AD 11 D0    LDA D011
,6013   29 7F       AND #7F
,6015   8D 11 D0    STA D011
,6018   A9 81       LDA #81
,601A   8D 1A D0    STA D01A
,601D   A9 00       LDA #00
,601F   8D 20 D0    STA D020
,6022   A9 04       LDA #04
,6024   85 02       STA   02
,6026   58          CLI
,6027   60          RTS
-------------------------------
,6028   AD 19 D0    LDA D019
,602B   8D 19 D0    STA D019
,602E   30 07       BMI 6037
,6030   AD 0D DC    LDA DC0D
,6033   58          CLI
,6034   4C 31 EA    JMP EA31
-------------------------------
,6037   AD 12 D0    LDA D012
,603A   C9 F8       CMP #F8
,603C   B0 11       BCS 604F
,603E   18          CLC
,603F   65 02       ADC 02
,6041   8D 12 D0    STA D012
,6044   A0 03       LDY #03
,6046   88          DEY
,6047   D0 FD       BNE 6046
,6049   EE 20 D0    INC D020
,604C   4C 81 EA    JMP EA81
-------------------------------
,604F   A9 00       LDA #00
,6051   8D 20 D0    STA D020
,6054   A9 32       LDA #32
,6056   8D 12 D0    STA D012
,6059   4C 81 EA    JMP EA81
-------------------------------
,605C   78          SEI
,605D   A9 00       LDA #00
,605F   8D 1A D0    STA D01A
,6062   A9 31       LDA #31
,6064   8D 14 03    STA 0314
,6067   A9 EA       LDA #EA
,6069   8D 15 03    STA 0315
,606C   A9 0E       LDA #0E
,606E   8D 20 D0    STA D020
,6071   58          CLI
,6072   60          RTS
-------------------------------
```

Assembler ist keine Alchemie
Folge 13

Erschienen in: 64'er 10/1985
Verantwortlicher Redakteur: Georg Klinge
Autor: Heimo Ponnath

Im Gegensatz zum sonstigen Sprachgebrauch erregt das Kürzel »CIA« bei Commodore-64-Kennern angenehme Assoziationen.

Die beiden CIAs (Complex Interface Adapter) unseres Computers und ihre Rolle bei der Unterbrechungs-Programmierung sollen in dieser Folge entschleiert werden.

Lassen Sie uns kurz rekapitulieren: Als primäre Unterbrechungsanforderer hatten wir drei Bausteine unseres Computers benannt, nämlich den VIC-II-Chip und die beiden CIA-Bausteine. CIA kommt von »Complex Interface Adapter« und ist die Bezeichnung für die beiden Ein- und Ausgabe-Bausteine, die den gesamten Verkehr zwischen dem zentralen Gehirn unseres C 64 und der Peripherie managen. Wir hatten bemerkt, dass ein CIA, der IRQ-CIA (Adressen von 56 320 bis 56 575), ausschließlich für die maskierbaren Unterbrechungen zuständig ist. Dazu gehören die 60-mal pro Sekunde stattfindenden »Timer-Interrupts«, welche die Cursorbehandlung, die TI$-Uhr, die Tastaturabfrage etc. bearbeiten. Der andere CIA, genannt NMI-CIA (Adressenraum 56 576 - 56831), ist nur für die nicht maskierbaren Unterbrechungen verantwortlich und wird bei normaler Nutzung des C 64 so gut wie nie eingesetzt. Ich gehe im Folgenden davon aus, dass Sie keine RS232C-Schnittstelle in Ihren Computer eingesetzt haben. Sollte das aber der Fall sein, dann müssten Sie darauf achten, die folgenden Beispiele - die den NMI-CIA betreffen - ohne gleichzeitigen Betrieb dieser

Register-Nr. ($)	Adresse (dez.)		Name	Funktion
	CIA 1	CIA 2		
04	56324	56580	TALO	TIMER A (LSB)
05	56325	56581	TAHI	TIMER A (MSB)
06	56326	56582	TBLO	TIMER B (LSB)
07	56327	56583	TBHI	TIMER B (MSB)
08	56328	56584	TOD10THS	$^1/_{10}$-Sekunden-Register
09	56329	56585	TODSEC	Sekunden-Register
0A	56330	56586	TODMIN	Minuten-Register
0B	56331	56587	TODHR	Stunden-Register, AM/PM-Flagge
0D	56333	56589	JCR	Unterbrechungs-Kontrollregister
0E	56334	56590	CRA	Kontrollregister A
0F	56335	56591	CRB	Kontrollregister B

Tab. 30: Die wichtigen Register der beiden CIAs

Schnittstelle anzuwenden, weil sich sonst Störungen ergeben könnten.

In Folge 10 haben wir uns ein Register (das Register 13, Interrupt-Kontrollregister) der CIAs schon genauer angesehen und auch die Unterschiede beider Bausteine festgestellt. Dort war die Rede von Timern, Echtzeituhren, Alarm-Funktionen etc. Was es damit auf sich hat und wie man diese Möglichkeiten nutzen kann, soll diesmal unser Thema sein. Wir werden uns dazu alle Register der CIAs genauer ansehen, die für die von uns ausgewählten Unterbrechungsoptionen eine Rolle spielen.

Dabei fallen einige unter den Tisch - das habe ich aber schon in Folge 10 angekündigt -, nämlich diejenigen, die mit dem Verkehr über den seriellen Port beziehungsweise über die RS232C-Schnittstelle zu tun haben.

In Tabelle 30 finden Sie zunächst eine Übersicht der von uns behandelten Register.

Sie sehen, dass jeder CIA über zwei so genannte Timer (A und B) verfügt, sodann über die »Time of Day« (zu deutsch etwa »Tageszeit«) genannte Echtzeituhr mit vier Registern und schließlich noch über drei Kontrollregister, zu denen auch das schon erwähnte Register 13 gehört. Sehen wir uns zunächst die Timer an:

Die Timer der CIAs

Insgesamt verfügen wir über vier dieser Timer: Timer A und B im CIA 1 und dasselbe noch mal im CIA 2. Es handelt sich dabei um 16-Bit-Register, in die ein Startwert geschrieben werden kann, von dem an dann heruntergezählt wird. Jedes Mal, wenn dann der Wert 0 unterschritten ist, gibt es für uns die Möglichkeit, bestimmte Ereignisse stattfinden zu lassen. Man kann diese Register unabhängig voneinander, aber auch kombiniert benutzen.

Ein Lesen des Registers liefert immer den momentan gerade aktuellen Wert. Ein Schreiben in das Register führt automatisch zum Festlegen eines Startwerts. Was an Optionen mit diesen Timern möglich ist, wird über Kontrollregister gesteuert. Das CRA (Register $0E) bezieht sich vor allem auf den Timer A, das CRB (Register $0F) auf Timer B. Die 16-Bit-Register werden - wie gewohnt - in der Form LSB/MSB betrieben. In den Timer A des CIA 1 wird bei jedem I/O-Reset folgendes Wertepaar eingetragen:

56324	dezimal 37	LSB
56326	dezimal 64	MSB

Das entspricht einem Startwert von 16421. Im PAL-System hat der Quarz, der die Taktfrequenz bestimmt, eine Frequenz von 17,734472 MHz.

Die Prozessorfrequenz errechnet sich daraus mittels Division durch 18 zu 985248,4 Hz (also etwas weniger als 1 MHz, was den europäischen C64 langsamer macht als den amerikanischen, der etwas mehr als 1 MHz verwendet).

Wenn mit dieser Geschwindigkeit der Timer heruntergezählt wird, erhält man genau einen Unterlauf alle $^1/_{60}$ Sekunden. Das ist der Weg, eine kontrollierte Zeitspanne durch den Timer zählen zu lassen.

Sei X der gesuchte Startwert, der zu einer Spanne von T Sekunden führt, dann kann man X wie folgt berechnen:

$$X = 985248{,}4 * T$$

Der Integerwert von X ist dann in ein LSB und ein MSB zu teilen und in die Timer-Register einzutragen. Allerdings ergibt sich so eine natürliche Grenze:

Die höchste durch 2 Bytes darstellbare Zahl ist ja 65535. Wenn wir diesen Wert in den Timer schreiben, dann ist er alle $^1/_{15}$ Sekunden auf 0 heruntergezählt. Für längere Zeiten ist aber vorgesorgt. Die beiden Timer A und B sind zu einem 32-Bit-Register kombinierbar (wie, dazu kommen wir gleich noch). Die höchste Zahl X ist dann:

$$4294967296 = 2^{32}$$

Damit kann im Extremfall eine Herabzählzeit von 1 Stunde, 12 Minuten und zirka 40 Sekunden eingeplant werden, was für die meisten Zwecke ausreichen dürfte.

Möchten Sie also genau eine Sekunde Spielraum beim Herunterzählen haben, dann muss die Zahl 985 248 als 4-Byte-Integer-Wert in die Speicher von Timer A und Timer B gebracht werden. Das führt dann zu den Werten 0, 15, 8, 160 (weil 985 248 = 0 * 16777216 + 15 * 65 536 + 8 * 256 + 160).

0 und 15 gelangen als MSB beziehungsweise LSB in Timer B (also Register 07 und 06), 8 und 160 sind MSB und LSB für den Timer A (Register 05 und 04).

Sehen wir uns nun an, wie wir dem Computer sagen, was mit diesen Startwerten in den Timer-Registern geschehen soll. Die beiden Kontrollregister CRA und CRB beziehen sich weitgehend auf die gleichnamigen Timer. In Abbildung 37 finden Sie das Register $0E, also CRA, und in Abbildung 38 das andere Kontrollregister CRB ($0F).

Die Bedeutung der Bits 0 bis 4 ist - jeweils für den dazugehörigen Timer - identisch:

Bit 0
an dieser Stelle führt zum sofortigen Anhalten des Timers. 1 in diesem Bit startet das Herunterzählen.

Bits 1 und 2
Diese beiden Bits hängen mit dem externen Signalverkehr zusammen und sollen für uns außer Acht bleiben.

Bit 3
Ist dieses Bit = 1, dann ist der so genannte »One Shot«-Betrieb des Timers aktiv. Das bedeutet, dass vom Startwert an bis auf Null heruntergezählt wird.

Dann findet das programmierte Ereignis statt (zum Beispiel ein IRQ). Anschließend wird der Startwert wieder eingeladen und der Timer gestoppt.

Im Gegensatz dazu läuft der »Continuous«-Betrieb, wenn das Bit den Wert 0 enthält. Dabei geschieht zunächst dasselbe wie beim »One Shot«-Modus, der Timer wird aber nicht angehalten, sondern der ganze Vorgang wiederholt sich in einer Endlosschleife.

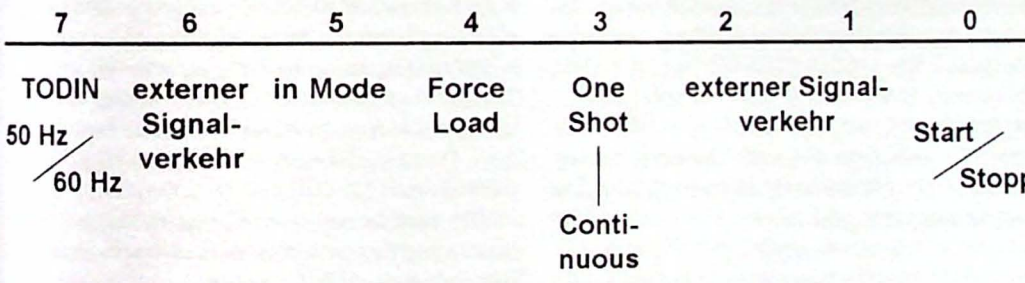


Abb. 37: Das Kontrollregister von Timer A

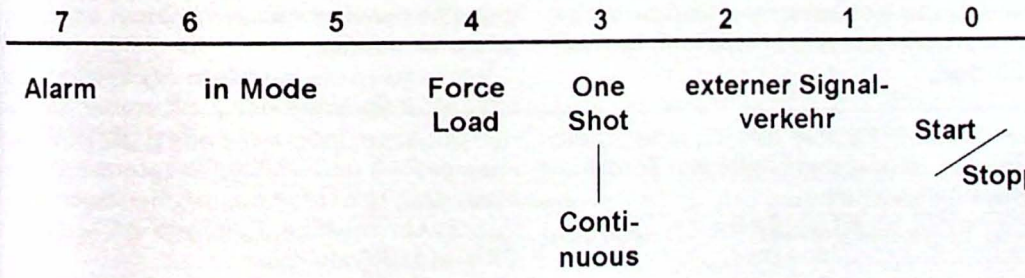


Abb. 38: Das Kontrollregister von Timer B

Bit 4
Das Hineinschreiben einer 1 in dieses Bit erzeugt ein sofortiges Neuladen der Timer-Register mit dem Startwert. Dabei ist es gleichgültig, ob der Timer gerade läuft oder nicht. Schreibt man eine Null hinein, hat das keine Wirkung. Beim Lesen des Registers ist dieses Bit immer 0. Zu diesem Bit und seiner Wirkung ist noch etwas zu sagen: Das Neuladen des Timers geschieht

- immer dann, wenn ein Unterlauf stattgefunden hat
- falls der Timer steht und in die Register ein Startwert geschrieben wird. Dabei ist der CIA so konstruiert, dass man kein zwangsweises Laden (also mit Bit 4 = 1) braucht, wenn man den Startwert in der Reihenfolge LSB/MSB in die Register bringt.

Die Bits 5 bis 7 haben nun unterschiedliche Bedeutung im CRA und im CRB: Register CRA ($0E)

Bit 5:
Ist dieses Bit gleich Null, dann wird im Systemtakt gezählt. Den hatten wir vorhin zur Zeitberechnung schon verwendet. Wenn das Bit auf 1 gesetzt ist, zählt der Timer externe Signale.

Bit 6:
Spielt für den Signalverkehr über den seriellen Port eine Rolle und soll uns hier nicht weiter beschäftigen.

Bit 7:
Damit steuert man nicht den Timer A, sondern dieses Bit bezieht sich auf die gleich noch zu behandelnde Echtzeituhr. Register CRB ($0F).

Die Bits 5 und 6 sind hier im Zusammenhang von Bedeutung. Es gibt vier Kombinationsmöglichkeiten:

Bit 6 — Bit 5
0 — 0

Der Timer B wird - wie vorhin der Timer A - im Systemtakt heruntergezählt.

Bit 6 — Bit 5
0 — 1

Der Timer B wird durch externe Signale heruntergezählt.

Bit 6 — Bit 5
1 — 0

Der Timer B zählt die Unterläufe von Timer A. Das ist der vorhin erwähnte Punkt, der beide Timer zum 32-Bit-Zähler kombiniert. Man kann also im Extremfall 65536-mal 68536 Takte zählen lassen.

Bit 6 — Bit 5
1 — 1

Auch in diesem Fall zählt Timer B die Unterläufe von Timer A. Er tut das aber nur, wenn ein bestimmtes externes Signal vorhanden ist.

Auch beim Register CRB steuert Bit 7 bestimmte Möglichkeiten der Echtzeituhr. Deshalb haben Sie bitte noch ein wenig Geduld, bis wir diese Uhr behandeln.

Wir wissen nun ganz gut, wie wir mit den Timern umzugehen haben. Unser Wissen wollen wir nun in einem kleinen Test erproben. Dazu bedienen wir uns des $^1/_{60}$-Sekunden-IRQs.

Wir verändern diese regelmäßige Unterbrechung derart, dass sie nur noch einmal in der Sekunde stattfindet.

Welche Zahlen dazu in ein 32-Bit-Register gepackt werden müssen, haben wir eben schon berechnet.

Wir müssen sie jeweils in der Reihenfolge LSB/MSB einschreiben und vorher die Timer anhalten, indem die Bits 0 der Kontrollregister CRA und CRB auf 0 gesetzt werden. Nach dem Einschreiben und Starten der beiden Timer müssen folgende Bitmuster in CRA und CRB stehen:

CRA
Bit 0 = 1 Start Timer A
Bit 3 = 0 Dauerlauf
Bit 5 = 0 Systemtakt

CRB
Bit 0 = 1 Start Timer B
Bit 3 = 0 Dauerlauf
Bit 5 = 0
Bit 6 = 1 Timer B zählt Unterläufe von Timer A.

Bevor wir die Timer starten, muss auch noch das Interrupt-Kontrollregister verändert werden (das hatten wir uns in der 10. Folge genauer angesehen). Bislang erzeugt ein Unterlauf des Timers A eine Unterbrechung. Wir möchten aber, dass der Timer B (damit wir das 32-Bit-Register voll ausnutzen) der Auslöser ist. Dazu muss Bit 0 des ICRs gelöscht und stattdessen Bit 1 gesetzt werden.

Im Programm »Timer-Test« (siehe Programme 8 und 9) ist all das realisiert. Mit `SYS 49152` gestartet, zeigt sich sofort ein deutlich verlangsamter Cursor. Noch langsamer kann alles werden, indem Sie höhere Werte in die Timer-Register schreiben. Den Normalzustand stellen Sie einfach durch Drücken der <RUN/STOP>- und der <RESTORE>-Tasten her. Dabei wird ja - wie Sie von der letzten Folge her wissen - auch ein I/O-Reset ausgeführt, der den Ausgangszustand wiederherstellt.

Die Verlängerung des IRQ-Zyklus' hat übrigens noch einen sinnvollen Nebeneffekt: Je seltener ein laufendes Programm unterbrochen wird, desto schneller wird es mit seinen Jobs fertig. Das kann man immer dann tun - im Extremfall sogar den IRQ ganz ausschalten - wenn man die Möglichkeiten, die der Computer während des normalen IRQs anbietet, nur selten oder aber gar nicht braucht.

Die Echtzeituhren

Wir kennen nun fünf Uhren in unserem Computer: die vier Timer (jeweils A und B im CIA 1 und CIA 2), die wir, weil wir die Impulszahlen in Zeiteinheiten umrechnen können, zur Zeitmessung einsetzen könnten und die im BASIC verfügbare Uhr TI$, die aber - wie wir nun wissen - lediglich die Umsetzung des Timers A im CIA 1 in ein bequemer handhabbares Software-Instrument ist.

Zudem ist die Ganggenauigkeit dieser Uhr recht gering. Schon einige Kassettenoperationen genügen, um sie völlig aus dem Takt zu bringen.

Um so mehr verwundert es, dass zwei hervorragende Echtzeituhren im Commodore 64 so gut wie nie benutzt werden, ja nicht einmal in irgendeiner Weise softwaremäßig unterstützt werden. Vielleicht ist das ein bisschen zuviel »mehr sein als scheinen«, das Commodore da betreibt - wenn man bedenkt, welche verborgenen Schätze da bei genauer Untersuchung des Computers alle zu Tage gefördert werden können (man denke nur an die hochauflösende Grafik).

Jeder der beiden CIAs verfügt über solch eine Uhr, die direkt von der Netzfrequenz getaktet wird. Die Zählung der Zeit geschieht in vier Registern (Register $08 bis $0B), die in Abbildung 39 gezeigt werden.

Vielleicht fällt Ihnen etwas auf, wenn Sie sich diese vier Bytes mal genauer ansehen: Die Speicherung geschieht in Form von Einer- und Zehnerstellen. Das kann also we-

Register Name	Nr.	7	6	5	4	3	2	1	0
TOD10THS	08	unbenutzt					1/10-Sekundenwert		
TODSEC	09	unbenutzt	Zehnerstelle Sekunden				Einerstelle Sekunden		
TODMIN	0A	unbenutzt	Zehnerstelle Minuten				Einerstelle Minuten		
TODHR	0B	AM/PM-Flagge	unbenutzt	Zehnerstelle Stunden			Einerstelle Sekunden		

Abb. 39: Die Register der Echtzeituhren

der im Binärformat noch als ASCII-Zeichen funktionieren. Hier werden die Ziffern als BCD-Zahlen abgelegt. In der dritten Folge wurde dieses »Binary Coded Decimal«-Format erklärt. Das ist lange her und soll deshalb hier nochmal vorgestellt werden, damit alle wissen, wovon die Rede ist:

In dieser Zahlendarstellung wird jede Dezimalstelle einer Zahl gesondert in eine Binärzahl umgewandelt. Dann ergibt sich der folgende Zusammenhang:

Binär	Dezimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9

Das war's! Die anderen möglichen Binärkombinationen (also zum Beispiel 1010 etc.) werden nicht benutzt. Die Zahl 25 beispielsweise lautet im BCD-Format:

```
0010 0101
  ^    ^
  2    5
```

Jetzt ist es Ihnen sicherlich verständlich, warum für die Sekunden- und Minuten-Zehnerstellen nicht mehr als drei Bits reserviert wurden: Größer als 6 wird die Zehnerstelle nicht.

Zum Stundenregister TODHR ist aber noch etwas zu sagen: Dort ist für die Stunden-Zehnerstelle nur ein Bit reserviert. Die Uhr läuft nicht bis 24 Uhr, sondern lediglich bis 12 Uhr. Zur Unterscheidung, ob Vor- oder Nachmittags gemeint ist, dient das Bit 7. Diese so genannte »AM/PM-Flagge« ist an der angelsächsischen Gewohnheit orientiert, zum Beispiel für 16 Uhr den Ausdruck »4 PM« zu verwenden. PM kommt vom lateinischen »Post Meridiem«, was übersetzt heißt »nach dem Mittag«, wohingegen AM für »Ante Meridiem« steht, also »vor dem Mittag«. Meint man nun AM, dann muss diese Flagge auf 0, bei PM aber auf 1 gesetzt sein.

Beim Stellen der Uhren sollte eine Reihenfolge eingehalten werden. Sobald nämlich in das Stundenregister geschrieben wird, hält die Zählung automatisch an. Man kann nun die anderen Werte in die Register schreiben. Den Startschuss liefert das Schreiben in das Register TOD10TH: Von nun an tickt die Uhr wieder.

Ähnlich funktioniert das Lesen der Uhrzeit. Sobald das Stundenregister gelesen wird, führt das zum Anhalten der Uhr, so dass die restlichen Register reibungslos auslesbar sind. Wieder ist es das Zehntelsekundenregister, das beim Auslesen ein Weiterlaufen der Uhr bewirkt. Aber - so werden Sie bemerken - wenn der Auslesevorgang eine bestimmte Zeit beansprucht, führt das doch zu Verzögerungen! Die Lösung ist, dass der gesamte Inhalt der vier Register gleichzeitig mit dem Auslesen des Stundenwerts in einen internen Speicher transferiert wird und dort weiterläuft. Nach dem Lesen des TOD10TH kommt der aktuelle Wert zurück in die Register und dieser wird weitergezählt.

Nun wird es höchste Zeit, dass wir uns die beiden Bits im CRA und im CRB ansehen, die wir vorhin bei der Timer-Behandlung links liegen ließen. Bit 7 im CRA kündigt der Echtzeituhr an, welche Netzfrequenz zu erwarten ist. Eine 1 an dieser Stelle steht für 50 Hz, eine 0 für 60 Hz. Das Stromnetz in Deutschland liefert einen Wechselstrom mit 50 Hz, weshalb wir dort die 1 setzen sollten.

Da gibt es ein kleines Problem: Beim I/O-Reset, der durch Drücken der <RUN/STOP>- und der <RESTORE>-Tasten zusammen ausgelöst wird, schreibt der Computer immer den amerikanischen Wert für 60 Hz in dieses Bit. Dann geht die Uhr aber empfindlich

nach. Man muss also einen Weg finden, der es erlaubt, dort in diesem Fall wieder eine 1 einzuschreiben. Das ist durch eine eigene NMI-Routine möglich. Sie sehen schon: Der Weg zur Nutzung dieser verlockenden Uhren ist ziemlich dornenreich!

Noch interessanter ist das Bit 7 im CRB. Das Setzen der Uhrzeit ist nämlich nur möglich, wenn dieses Bit den Inhalt 0 hat. Was geschieht, wenn dort eine 1 steht? Dann bestimmt man nicht die aktuelle Uhrzeit, sondern stellt einen Wecker (das ist die Alarmzeit). Das geschieht nach dem Setzen dieses Bits genauso wie vorhin das Einschreiben der Uhrzeit (also erstaunlicherweise auch in genau dieselben Register!). Im Unterschied dazu ist allerdings ein Lesen der Alarmzeit nicht möglich - das ergibt immer die aktuelle Uhrzeit. Man muss für diesen Fall die Weckzeit irgendwo abspeichern und bei Bedarf dann von dort lesen.

Weil man ja meistens nach dem Erreichen der Alarmzeit irgendeine Reaktion erwartet, ist im ICR (also dem Unterbrechungskontrollregister 13) jedes CIAs noch ein Bit reserviert - das Bit 2 -, mit dessen Hilfe der Alarm per IRQ oder NMI wie auch immer geartet losbrechen kann. Der Fantasie sind hier nur wenige Grenzen gesetzt. Wie man mit diesem ICR umgeht, ist Ihnen noch aus Folge 10 geläufig.

Damit sind wir durch die Eigenheiten der CIAs durch. Man braucht tatsächlich keine Scheu zu haben, diese Echtzeituhren zu nutzen. Lediglich die Uhr im CIA 1 wird manchmal verwendet, um einen bestimmten Wert für die Zufallszahlenerzeugung zu generieren. Aber das sollte einer eigenen Uhren-Routine nicht in die Quere kommen. Solch eine Echtzeituhr finden Sie in den Programmen 10 und 11.

Durch `SYS 49152` aktivieren Sie die Uhr, die Sie mit `SYS 49261` wieder abschalten können. Durch das USR-Kommando `A = USR (String)` stellen Sie die Startzeit ein. `String` kann dabei eine Stringvariable sein oder auch direkt ein String der Form »HH MMSST« (also Stunden, Minuten, Sekunden, Zehntelsekunden). In A steht eine 0, wenn kein Fehler, aber eine -1, wenn ein Fehler aufgetreten ist. Das Lesen der Uhr erfolgt über ein zweites USR-Kommando: `PRINT USR(Zahl)`. Dabei kann `Zahl` eine beliebige Zahl oder Variable sein. Eine Alarmzeit ist ebenfalls durch ein USR-Kommando einstellbar, in dem vor der Zeiteingabe noch ein Buchstabe steht. Beispielsweise stellt `A = USR("A1200000")` einen Wecker auf 12 Uhr.

Der Alarm im Programm lässt den Bildschirmrahmen blinken. Abstellen kann man das durch Auslösen eines RESTORE-NMI (also <RUN/STOP> und <RESTORE>). Sollten Sie vor dem eingestellten Alarm mal solch einen NMI auslösen, dann muss die Alarmzeit neu gestellt werden.

Als Basis für dieses Programm diente ein Listing aus dem schon oft erwähnten Buch von Babel / Krause / Dripke »Das Interface-Age-Systemhandbuch zum Commodore 64«.

Die Unterbrechungs-Programmierung ist damit abgeschlossen - ebenso diese Serie, die als Einführung in die Assembler-Alchemie nun alle Geheimnisse der Kunst aufgedeckt hat. In den letzten Folgen sind wir schon in die Meistergrade der Zunft aufgestiegen.

So wie die Segler sich oft »Mast- und Schotbruch« wünschen, verabschiede ich mich, indem ich Ihnen viele grandiose Abstürze wünsche.

Progr. 9: »Timer-Test«, ein Beispiel für die Anwendung eines 32-Bit-Timers

```
programm : prg.timer-testc000 c051
----------------------------------
c000 : 78 ad 0e dc 29 fe 8d 0e  4b
c008 : dc ad 0f dc 29 fe 8d 0f  f9
c010 : dc a9 0f 8d 06 dc a9 00  24
c018 : ad 07 dc a9 a0 8d 04 dc  d5
c020 : a9 08 8d 05 dc a9 1f 8d  84
c028 : 0d dc a9 82 8d 0d dc ad  6e
c030 : 0e dc 29 d7 8d 0e dc ad  0a
c038 : 0f dc 29 d7 8d 0f dc ad  1b
c040 : 0e dc 09 01 8d 0e dc ad  37
c048 : 0f dc 09 41 8d 0f dc 58  a5
c050 : 60 ff 00 ff 00 ff 00 ff  b0
```

Progr. 10: Eine Echtzeituhr

```
programm : obj.alarmuhr c000  c18d
----------------------------------
c000 : a9 8e 8d 11 03 a9 c0 8d  11
c008 : 12 03 a9 1d 8d 18 03 a9  a3
c010 : c0 8d 19 03 ad 0e dd 09  12
c018 : 80 8d 0e dd 60 48 8a 48  a1
c020 : 98 48 a9 7f 8d 0d dd ac  49
c028 : 0d dd 10 06 4c 6a c1 4c  a0
c030 : 72 fe 20 bc f6 20 e1 ff  b9
c038 : d0 f5 a2 04 bd 2f fd 9d  b4
c040 : 13 03 ca d0 f7 a2 1a bd  1a
c048 : 35 fd 9d 19 03 ca d0 f7  c0
c050 : a9 7f 8d 0d dc 8d 0d dd  e8
c058 : 8d 00 dc a9 08 8d 0e dc  30
c060 : a9 88 8d 0e dd a9 08 20  fe
c068 : b6 fd 4c 6c fe a9 48 8d  37
c070 : 11 03 a9 b2 8d 12 03 78  2a
c078 : a9 47 8d 18 03 a9 fe 8d  c0
c080 : 19 03 a9 31 ad 14 03 a9  84
c088 : ea 8d 15 03 58 60 24 0d  12
c090 : 30 03 4c 20 c1 20 82 b7  f0
c098 : c0 07 d0 40 ad 0f dd 29  35
c0a0 : 7f 8d 0f dd a0 00 a9 24  5e
c0a8 : 20 fe c0 d0 02 a9 24 c9  23
c0b0 : 13 90 07 f8 38 e9 12 d8  b9
c0b8 : 09 80 8d 0b dd 20 fc c0  1a
c0c0 : 8d 0a dd 20 fc c0 8d 09  ec
c0c8 : dd 20 66 c1 8d 08 dd a9  6b
c0d0 : 00 4c 3c bc 68 68 68 68  d9
c0d8 : a9 ff d0 f5 c0 08 d0 f8  f5
c0e0 : 8d 0f dd 09 80 8d 0f dd  1a
c0e8 : a9 84 8d 0d dd a9 3c 85  ff
c0f0 : 04 85 02 a9 ff 85 03 a0  b6
c0f8 : 01 4c a6 c0 a9 60 85 24  dd
c100 : 20 13 c1 08 0a 0a 0a 85  80
c108 : 25 20 13 c1 05 25 c5 24  13
c110 : b0 c0 60 b1 22 38 e9 30  5d
c118 : 90 ba c9 0a b0 b6 c8 60  5e
c120 : a9 07 20 7d b4 a0 00 ad  b0
c128 : 0b dd 08 29 1f c9 12 d0  73
c130 : 02 a9 00 28 10 05 f8 18  49
c138 : 69 12 d8 20 55 c1 ad 0a  13
c140 : dd 20 55 c1 8d 09 dd 20  96
c148 : 55 c1 8d 08 dd 20 60 c1  ce
c150 : 68 68 4c ca b4 48 4a 4a  a4
c158 : 4a 4a 20 60 c1 68 29 0f  fe
c160 : 09 30 91 62 c8 60 20 13  68
c168 : c1 60 a9 77 8d 14 03 a9  8b
c170 : c1 8d 15 03 4c bc fe c6  b2
c178 : 02 f0 03 4c 31 ea a5 04  46
c180 : 85 02 ad 20 d0 45 03 8d  d4
c188 : 20 d0 4c 31 ea 00 ff 00  f8
```

Hinweis:

Die Quellkodes zu den Programmen 9 und 10 können Sie als JPG-Dateien unter folgendem Link herunterladen:

Programm 9
http://www.skriptorium-vd.de/01/quellkode9.jpg

Programm 10
http://www.skriptorium-vd.de/01/quellkode10.jpg

Ohne gutes Werkzeug geht es nicht: SMON

Ohne gutes Werkzeug geht es nicht: SMON Folge 1

Erschienen in: 64'er 11/1984
Verantwortlicher Redakteur: Georg Klinge
Autoren: Dietrich Weineck und Norfried Mann

In mehreren Teilen möchten wir Ihnen einen Maschinensprachmonitor vorstellen. Parallel zum Kursus über Assembler-Programmierung wird Schritt für Schritt ein Programm entstehen, das sich durchaus mit kommerziellen Monitoren messen kann.

Ich kann mich noch gut an unsere ersten Schritte in die Maschinensprache erinnern. Ausgerüstet mit einer Befehlsliste für den 6502 und einem in BASIC geschriebenen »Mini-Monitor« entstanden Programme, die 3 und 5 addieren und das Ergebnis im Speicher ablegen konnten. Dazu mussten wir die Befehlkodes aus der Liste heraussuchen und dann in den Speicher »POKEn«. Jeder Sprung musste von Hand ausgerechnet werden, jeder falsch herausgesuchte Befehl führte zum Programmabsturz.

Der erste Disassembler -ein Programm zur Anzeige der Maschinenbefehle in Assemblersprache - war für uns die Offenbarung. Von nun an konnten wir Maschinenprogramme analysieren und daraus lernen.

Zum Verständnis der Maschinensprache ist es nämlich noch weit mehr als bei anderen Sprachen wichtig, vorhandene Programme zu verstehen und sich dabei die wichtigsten Techniken anzueignen.

Mit der Zeit wuchsen unsere Ansprüche: Ein Assembler musste her, um die neu gewonnenen Erkenntnisse auch auszuprobieren. Das war zuerst wieder ein BASIC-Programm, langsam und wenig komfortabel, aber immerhin.

Wir schrieben unsere ersten kleinen Routinen, vor allem, um vorhandene Maschinenprogramme unseren eigenen Wünschen anzupassen. Mit dem AMON für den VC 20 bekamen wir dann einen Monitor, der (fast) alle unsere Wünsche erfüllte. Als wir jedoch auf den C 64 umstiegen, mussten wir feststellen, dass es für diesen Computer nichts gab, das uns zufrieden stellen konnte. Der einzige Ausweg: selbst programmieren.

So entstand im Laufe eines Jahres SMON. Ursprünglich hatten wir nur vor, die Funktionen von AMON für den C 64 zu programmieren, aber dabei blieb es nicht. Immer neue Befehle und Routinen kamen hinzu, bis wir endlich zufrieden waren.

Was bietet SMON?

Zunächst ist alles enthalten, was zum »Standard« gehört: Memory-Dump, also die Anzeige des Speicherinhalts in Hex-Bytes, mit Änderungsmöglichkeiten, ein Disassembler mit Änderungsmöglichkeit sowie Routinen zum Laden, Abspeichern und Starten von Maschinenprogrammen.

Darüber hinaus gibt es einen kleinen Direktassembler, der sogar Labels verarbeitet, Befehle zum Verschieben im Speicher mit und ohne Umrechnen der Adressen und Routinen zum Umrechnen von Hex-, Dezimal- und Binärzahlen. Der besondere Clou von SMON liegt aber zweifellos in seinen leis-

tungsfähigen Suchroutinen und vor allem im Trace-Modus. Damit lassen sich Maschinenprogramme Schritt für Schritt abarbeiten und kontrollieren.

Dieser erste Teil umfasst sämtliche Eingabe- und Ausgaberoutinen, die Registeranzeige, den Memory-Dump sowie Disassembler und Assembler. Damit steht Ihnen bereits ein lauffähiges Monitorprogramm mit den unten aufgeführten Befehlen zur Verfügung.

Der Monitor benötigt für alle Eingaben die hexadezimale Schreibweise, das heißt zu den Zahlen 1 bis 9 kommen noch die Buchstaben A (für dez. 10) bis F (für dez. 15) hinzu.

Bei der Eingabe von Adressen ist Folgendes zu beachten: [ANFADR] bedeutet exakt die Startadresse, [ENDADR] bedeutet die erste Adresse hinter dem gewählten Bereich. Im Normalfall ist die Eingabe mit und ohne Leerzeichen zulässig. Beim Abweichen von dieser Regel wird darauf besonders verwiesen.

Assemblieren

A [ANFADR]

Assemblierung beginnt bei angegebener Adresse. Beispiel:

A 4000 Beginn bei Startadresse $4000

Nach Eingabe von »RETURN« erscheint auf dem Bildschirm die gewählte Adresse mit einem blinkenden Cursor. Die Befehle werden so eingegeben, wie sie der Disassembler zeigt: LDY #00 oder LDA 400E,Y und so weiter. »RETURN« schließt die Eingabe der Zeile ab.

Bei fehlerhafter Eingabe springt der Cursor wieder in die Anfangsposition zurück, ansonsten wird der Befehl disassembliert und nach Ausgabe der Hex-Bytes gelistet.

Zur Korrektur vorhergehender Zeilen gehen Sie mit dem Cursor zur Anfangsposition (hinter die Adresse) zurück, schreiben den Befehl neu und gehen nach »RETURN« mit dem Cursor wieder in die letzte Zeile. Falls Ihnen bei Sprüngen (Branch-Befehl, JSR und JMP) die Zieladressen noch nicht bekannt sind, geben Sie einfach so genannte »Label« ein.

Ein Label besteht aus dem Buchstaben »M« (für Marke) und einer zweistelligen Hex-Zahl von 01 bis 30. Zum Beispiel: BCC M01

Wenn Sie die Zieladresse für den Sprung erreicht haben, kennzeichnen Sie diese mit eben dieser »Marke«. Zum Beispiel: M01 LDY #00

Einzelne Bytes nimmt der Assembler an, indem Sie diese mit einem Punkt kennzeichnen: .00 oder .AB. In diesem Modus werden die Eingaben natürlich nicht disassembliert.

Nach Beendigung des Assemblierens geben Sie »F« ein. Danach sehen Sie alle Ihre Eingaben noch einmal aufgelistet und korrigieren bei Bedarf wie beim Disassembler (!) angegeben. Probieren Sie einmal das folgende Beispiel: A 4000

Der Assembler meldet sich mit: »4000« und einem blinkenden Cursor. Geben Sie nun ein (die Adressen erscheinen automatisch):

```
4000 LDY #00      4009 CPY #12
4002 LDA 400E,Y   400B BCC 4002
4005 JSR FFD2     400D BRK
4008 INY
```

Die folgenden Bytes werden wie beschrieben mit einem Punkt eingegeben. Sie werden nicht disassembliert.

```
400E.0D           4017.54
400F.0D           4018.20
4010.53           4019.53
4011.4D           401A.55
4012.4F           401B.50
4013.4E           401C.45
4014.20           401D.52
4015.49           401E.0D
4016.53           401F.0D
```

Drücken Sie anschließend »F«. Ihr Programm wird noch mal aufgelistet. Starten Sie es nun

mit »G 4000«. Es erscheint ein Text auf dem Bildschirm — lassen Sie sich überraschen.

Disassemblieren
D [ANFADR, ENDADR]
disassembliert den Bereich von ANFADR bis ENDADR, wobei ENDADR nicht eingegeben werden muss. Wird keine Endadresse eingegeben, erscheint zunächst nur eine Zeile:

```
ADR      HEXBYTES      BEFEHL
4000     A000          LDY #00
```

Mit der <SPACE>-Taste wird der jeweils nächste Befehl in der gleichen Art und Weise gezeigt. Wünschen Sie eine fortlaufende Ausgabe, drücken Sie <RETURN>. Die Ausgabe wird dann so lange fortgesetzt, bis eine weitere Taste gedrückt wird oder bis ENDADR erreicht ist. Mit <RUN / STOP> springen Sie jederzeit in den Eingabemodus zurück.

Das Komma, das vor der Adresse auf dem Bildschirm erscheint, ist ein »hidden command« (verstecktes Kommando). Es braucht nicht eingegeben zu werden, da es automatisch beim Disassemblieren angezeigt wird. So ermöglicht es ein einfaches Ändern des Programms. Fahren Sie mit dem Cursor auf den zu ändernden Befehl und überschreiben Sie ihn mit dem Neuen. Wenn Sie jetzt <RETURN> drücken, erkennt SMON das Komma als Befehl und führt ihn im Speicher aus. Achten Sie aber darauf, dass der neue Befehl die gleiche Länge (in Bytes) hat, und füllen Sie gegebenenfalls mit »NOPs« auf. Zur Kontrolle können Sie den geänderten Bereich noch einmal disassemblieren.

Lassen Sie als Beispiel einmal das Programm (siehe Befehl »A«) ab 4000 disassemblieren (»D 4000 4011«). Ändern Sie nun den ersten Befehl auf LDY #01. Die Änderung zeigt sich daran, dass die HEX-Bytes automatisch den neuen Wert annehmen. Starten Sie nun das Programm nochmals mit »G 4000«. Jetzt erscheint der Text mit nur einer Zeile Abstand auf dem Bildschirm.

Starten eines Maschinenprogramms (Go)
G [ADRESSE]
startet ein Maschinenprogramm, das bei ADRESSE beginnt. Das Programm muss mit einem BRK-Befehl abgeschlossen werden, damit ein Rücksprung in SMON erfolgen kann. Wird nach »G« keine Adresse eingegeben, benutzt SMON die, die mit dem letzten BRK erreicht worden ist und bei der Register-Ausgabe als PC auftaucht. Mit dem »R«-Befehl (siehe unten) werden die Register vorher auf gewünschte Werte gesetzt.

Memory-Dump
M [ANFADR ENDADR]
gibt die HEX-Werte des Speichers sowie die zugehörigen ASCII-Zeichen aus. Auch hier kann auf die Eingabe einer Endadresse verzichtet werden. Die Steuerung der Ausgabe entspricht der beim Disassemblieren.

Beispiel: M 4000 gibt die Inhalte der Speicherstellen $4000 bis $4007 aus. Weiter geht es wie beim Disassemblieren mit <SPACE> oder <RETURN>.

Die Bytes können ebenfalls durch Überschreiben geändert werden, allerdings nicht die ASCII-Zeichen. Verantwortlich dafür ist der Doppelpunkt, der am Anfang jeder Zeile ausgegeben wird - ein weiterer »hidden command«. Wenn Ihre Änderung nicht durchgeführt werden kann, weil Sie zum Beispiel versuchen, ins ROM zu schreiben, wird ein »?« als Fehlermeldung ausgegeben.

Registeranzeige
R
zeigt den gegenwärtigen Stand der wichtigsten 6510-Register an: Programmzähler (PC), Status-Register (SR), Akkumulator (AC), X-Register (XR), Y-Register (YR) und Stack Pointer (SP).

Außerdem werden die einzelnen Flags des Status-Registers mit 1 für »gesetzt« und 0 für »nicht gesetzt« angezeigt. Durch Überschreiben werden die Inhalte auf einen gewünschten Wert gesetzt. Die Flags können allerdings nicht einzeln verändert werden,

sondern nur durch Überschreiben des Wertes von SR.

Exit

X

springt ins BASIC zurück. Alle BASIC-Pointer bleiben erhalten. Sie können also zum Beispiel direkt im Programm fortfahren, wenn Sie zwischendurch mit SMON einige Speicherstellen kontrolliert haben.

Probieren Sie alle bisher beschriebenen Befehle in Ruhe aus und machen Sie sich mit SMON vertraut. Arbeiten Sie auch parallel den Kurs über Assemblerprogrammierung in dieser Ausgabe durch. Alle Beispiele dort sind auf SMON abgestimmt.

Das Programm

Wir wollen jetzt einen Blick auf das Programm selbst werfen. Natürlich ist es unmöglich, den gesamten Quelltext umfassend zu beschreiben. Andererseits enthält SMON aber eine Reihe von Routinen, die in jedem Maschinenprogramm vorkommen. Wir werden im Rahmen dieser Serie versuchen, die wichtigsten zu erklären, damit Sie sie später in eigene Programme einbauen können.

Zum besseren Verständnis werden solche Routinen so abgedruckt, wie wir sie im Assembler-Quelltext geschrieben haben. Sie enthalten daher an Stelle absoluter Adressen Labels, deren Name - hoffentlich - etwas über den Sinn und Zweck aussagt. Parallel dazu sollten Sie sich diese Routinen von SMON disassemblieren lassen, damit Sie sehen, wie es denn nun fertig im Speicher aussieht.

Beginnen wir mit der Routine GETCHR-ERR. Das soll soviel bedeuten wie »Hole ein Zeichen und erzeuge eine Fehlermeldung, wenn keins eingegeben wurde«.

Leider wäre so ein Label auch für den geduldigsten Assembler viel zu lang, daher die merkwürdige Abkürzung. Mit dieser Routine holen wir ein Zeichen von der Tastatur. Das erledigt die Betriebssystemroutine CHRIN. Um zu prüfen, ob überhaupt etwas eingegeben wurde, untersuchen wir das Zeichen.

Handelt es sich um die <RETURN>-Taste ($0D), hat der Benutzer gar kein Zeichen eingegeben. Dies quittiert SMON mit einem »?« und dem Rücksprung in den Eingabemodus. So lässt sich - in gewissen Grenzen - kontrollieren, ob zu einem Befehl die richtigen Eingaben gemacht wurden. Geben Sie einmal den »D«-Befehl ohne Angabe einer Adresse ein, dann sehen Sie, was gemeint ist.

Alle Eingaberoutinen benutzen GETCHR-ERR, um Falscheingaben zu prüfen. Nehmen wir als Beispiel GETBYT. Diese soll ein Byte, also zwei ASCII-Zeichen 0 - F von der Tastatur holen und in ein Byte umwandeln.

Das erste Zeichen wird darauf überprüft, ob es sich um ein <SPACE> oder ein Komma handelt. Trifft das zu, wird es einfach übergangen und das nächste Zeichen geholt.

Der Benutzer kann also Leerzeichen und Komma benutzen, um seine Eingaben übersichtlicher zu machen, er muss aber nicht! Ist das Zeichen gültig, wird es von ASCHEX in eine Hexzahl gewandelt.

Dazu ein Beispiel: Auf der Tastatur wurde 5B eingetippt. Zuerst wird jetzt die 5 (ASCII $35) mit $3A verglichen, um festzustellen, ob es sich um eine Zahl (0 - 9) oder einen Buchstaben (A - F) handelt.

ASCII $35 ist eine Zahl, also wird nur die linke Hälfte ausmaskiert (AND #$0F). Ergebnis ist $05. Jetzt wird viermal nach links geschoben und das Ergebnis ($50) in $B4 zwischengespeichert.

Nun ist das B (ASCII $42) an der Reihe. Da $42 größer ist als $3A, werden diesmal 8 und das gesetzte Carry-Flag, also 9 addiert. Ergebnis ist $5B. Linke Hälfte ausmaskieren wie gehabt und eine OR-Verknüpfung mit dem gemerkten $50 ergibt $5B. Das war's.

Meistens aber braucht SMON zwei Bytes als Eingabe, zum Beispiel für Adressen. Mit dem, was wir schon haben, kein Problem: GETADR ruft einfach GETBYT zweimal hintereinander auf und legt das Ergebnis in zwei

Speicherstellen in der Zeropage ab, die mit dem X-Register ausgewählt werden können.

Brauchen wir mehr als eine Adresseingabe, rufen wir einfach GETADR mehrmals auf. So etwas machen GET3ADR und GET2ADR. Bisweilen aber, zum Beispiel beim »G«-Befehl, darf eine Adresse eingegeben werden, es muss aber nicht sein.

Deswegen prüft GETSTART, ob direkt nach dem »G« <RETURN> gedrückt wurde. Dies erledigt GETRET. Wenn ja, wird die Adresse benutzt, die in PCL und PCH steht. Das sind SMONs interne Programm-Counter. Ansonsten wird die eingetippte Adresse benutzt.

Sie sehen, wie aus einfachen Routinen immer kompliziertere Befehle zusammengesetzt werden.

Und das ist das ganze Geheimnis, wenn Sie umfangreiche Programme schreiben: Gliedern Sie sich das Problem (hier eine benutzerfreundliche Eingabe) in kleine und kleinste Schritte auf, die Sie dann jeden für sich programmieren und austesten.

Werfen wir noch einen Blick auf die Art und Weise, wie SMON Befehle verarbeitet. In EXECUTE setzen wir zunächst den Stackpointer auf den Wert, den er beim letzten BRK erreicht hatte. Dann werden als Erstes die »hidden commands« abgeprüft.

Wir lesen dazu direkt vom Bildschirm. D3 enthält die Anfangsadresse der aktuellen Zeile im Speicher. Übrigens gibt es neben den bereits erwähnten noch weitere »hidden commands«, die in den späteren Folgen noch auftauchen werden. Liegt kein verstecktes Kommando vor, holen wir mit GETCHRERR ein Zeichen und merken es uns in COMMAND. Jetzt untersuchen wir, ob dieses Zeichen in der Befehlsliste (CMDTBL) steht.

CMDTBL steht übrigens ab $C00B ganz oben im Speicher. Sie endet mit fünf Nullen für spätere Erweiterungen. Direkt dahinter stehen die Anfangsadressen der zugehörigen Routinen in der für den 6502 typischen Reihenfolge: Low-Byte zuerst, dann High-Byte. Sehen Sie sich das mit M C00B einmal an. Am Ende dieser Tabelle stehen nochmals 10 Nullen, denn zu jedem Byte in CMDTBL gehören ja zwei Adressbytes in der Liste (CMDS).

Wenn nun ein Kommando in CMDSEARCH gefunden wurde, wird CMDEXEC als Subroutine aufgerufen. CMDEXEC legt nun die zugehörigen Adressbytes auf den Stack und führt dann einen RTS aus, der jetzt - nach der Stackmanipulation - zu dem gewünschten Befehl führt.

Beachten Sie, dass RTS immer auf die um eins erhöhte Adresse springt, daher müssen Sie zu den Adressen in CMDS immer 1 addieren, wenn Sie den Anfang einer Routine suchen.

Alle Befehle in SMON enden mit einem RTS, springen also auf den JMP EXECUTE hinter CMDFOUND. Damit ist eine Endlosschleife geschlossen, die immer einen Befehl ausführt und anschließend wieder in die Eingabe zurückspringt.

Beim nächsten Mal erfahren Sie etwas über LOAD, SAVE und die Umrechnung verschiedener Zahlensysteme.

Ohne gutes Werkzeug geht es nicht: SMON Folge 2

Erschienen in: 64'er 12/1984
Verantwortlicher Redakteur: Georg Klinge
Autoren: Dietrich Weineck und Norfried Mann

Der Maschinensprache-Monitor SMON wird immer leistungsfähiger. Dieser zweite Teil erweitert ihn um wichtige Ausgabe-Routinen, lässt das Verschieben eines Programms mit und ohne Adressumrechnung zu und kann Zahlen vom Dezimal- in das Binärsystem und umgekehrt umrechnen.

Wir hoffen, dass wir Ihnen in der letzten Ausgabe nicht zuviel zugemutet haben und dass sich Ihre wunden Finger inzwischen wieder erholen konnten.

Bestimmt haben Sie im vergangenen Monat schon eifrig mit dem neuen Monitor gearbeitet und sind inzwischen mit den bisherigen Befehlen vertraut.

Denn nun folgt der zweite Teil und mit diesem natürlich wieder einige neue Befehle, die es zu erklären gilt.

Und das bieten wir Ihnen heute: I/O-SET, LOAD, SAVE, PRINTER-SET, die verschiedenen Zahlenumrechnungen (HEX-DEZ-BIN-ADD-SUB), OCCUPY, CONVERT, VERSCHIEBEN und WRITE.

I/O-SET

```
I 01
```

legt die Device-Nummer für LOAD und SAVE auf 1 (Kassette). Jedes Laden und Abspeichern erfolgt jetzt auf das angegebene Gerät. Die voreingestellte Device-Nummer ist 8 (für die Floppy also: `I 08`). Wenn Sie nur mit der Floppy arbeiten, brauchen Sie diesen Befehl also nicht.

LOAD

```
L "Name"
```

lädt ein Programm vom angegebenen Gerät (wie oben beschrieben) an die Originaladresse in den Speicher. Die BASIC-Zeiger bleiben von diesem Ladevorgang unbeeinflusst, das heißt, sie werden nicht verändert.

Beispiel: Unser Monitor soll an seiner Originaladresse ($C000) im Speicher stehen. Also brauchen Sie ihn nur mit `L "SMON"` zu laden, damit er dort erscheint.

Wenn Sie einmal ein Programm an eine andere als die Originaladresse laden wollen, dann bietet Ihnen SMON dazu folgende Möglichkeit: »`'L "Name" ADRESSE`« lädt ein Programm an die angegebene Adresse.

Nehmen Sie doch bitte noch einmal unser letztes Test-Programm und geben es mit dem Assembler ab Adresse $4000 ein. Speichern Sie es mit »`'S "SUPERTEST" 4000 4023`« ab und laden es dann

1. an die Originaladresse (`L "SUPERTEST"`) und
2. an eine andere Adresse (mit `L "SUPERTEST" 5000` zum Beispiel nach $5000).

Schauen Sie sich danach mit dem Disassembler-Befehl beide Routinen einmal an. Sie werden feststellen, dass beide Programme zwar bis auf die BRANCH-Befehle gleich aussehen, dass das Programm in $5000 aber nicht funktionieren kann, da es eine falsche Adresse verwendet (`5002 LDA 400E,Y`).

Ein anderes Beispiel dazu: Ein Autostart-Programm beginnt bei $0120, lässt sich aber in diesem Bereich nicht untersuchen, da dort der Prozessor-STACK (im Bereich von $0100 bis $01FF) liegt, der vom Prozessor selbst ständig verändert wird. Wenn Sie nun `L"Name"4120` eingeben, befindet sich das Pro-gramm anschließend bei $4120 (nicht an der Originaladresse $0120), und Sie können es ohne Einschränkungen – von den falschen Absolut-Adressen abgesehen – disassemblieren.

SAVE

`S "Name", ANFADR ENDADR`

speichert ein Programm von ANFADR bis ENDADR – 1 unter »Name« auf die Floppy ab, da diese – wie wir ja inzwischen wissen – das voreingestellte Gerät ist. Wenn Sie auf Kassette abspeichern wollen, setzen Sie vorher mit »`I 01`« die Device-Nummer auf 1.

Beispiel: `S "SUPERTEST" 4000 4020` speichert das Programm mit dem Namen »SUPERTEST« (es steht im Speicher von $4000 bis $401F) auf Diskette ab. Bitte beachten Sie auch bei diesem Befehl, dass die Endadresse auf das nächste Byte hinter dem Programm gesetzt wird.

PRINTER-SET

`P 02`

setzt die Primäradresse für den Drucker auf 2. Voreingestellt ist hier die 4 als Gerätenummer (zum Beispiel für Commodore-Drucker).

Vielleicht haben Sie es ja schon bemerkt: Bei allen Ausgabe-Befehlen (wie `D`, `M` etc.) können Sie auch den Drucker ansprechen, wenn Sie das Kommando geshiftet eingeben. Die Ausgabe erfolgt dann gleichzeitig auf Bildschirm und Drucker. (Beachten Sie bitte die Änderung für die Druckerausgabe am Schluss des Artikels.)

Ein bisschen Rechnerei

Die folgende Befehlsgruppe enthält Befehle zur Zahlenumrechnung. Sie wissen ja: Der Mensch mit seinen zehn Fingern neigt eher zur dezimalen Rechenweise, aber der Computer bevorzugt das Binärsystem, weil er nur zwei Finger hat (siehe Netzstecker).

Ein Kompromiss ist das Hexadezimalsystem, denn das versteht keiner von Beiden. Um Verständnisschwierigkeiten mit Ihrem Liebling aus dem Weg zu gehen, haben Sie aber SMON.

UMRECHNUNG DEZ-HEX

`#`

(Dezimalzahl) rechnet die Dezimalzahl in die entsprechende Hexadezimalzahl um. Hierbei können Sie die Eingabe in beliebiger Weise vornehmen, da SMON Zahlen bis 65535 umrechnet. Beispiel: `#12`, `#144`, `#3456`, `#65533` und so weiter.

UMRECHNUNG HEX-DEZ

`$`

(Hexadezimalzahl) rechnet die Hexadezimalzahl in die entsprechende Dezimalzahl um. Die Eingabe muss hierbei zweistellig beziehungsweise vierstellig erfolgen. Ist diese Zahl kleiner als $100 (= 255), wird zusätzlich auch der Binärwert ausgegeben.

Beispiel: `$12`, `$0012`, `$0D`, `$FFD2` etc. In den ersten drei Beispielen erfolgt die Anzeige auch in binärer Form.

UMRECHNUNG BINÄR-HEX, BINÄR-DEZ

`%`

(Binärzahl (achtstellig) rechnet die Binärzahl in die entsprechenden Hexadezimal- und Dezimalzahlen um. Bei diesem Befehl müssen Sie genau acht Binärzahlen eingeben.

Falls Sie einmal versehentlich mehr eingeben sollten, werden nur die ersten acht zur Umrechnung herangezogen.

Beispiel: `%00011111`, `%10101011`

ADD, SUB

`? 2340+156D`

berechnet die Summe der beiden vier(!)-stelligen Hex-Zahlen. Neben der Addition ist auch Subtraktion möglich.

Programme auf dem Rangierbahnhof

OCCUPY (Besetzen)

`O (ANFADR ENDADR HEX-Wert)`
belegt den angebenen Bereich mit dem vorgegebenen HEX-Wert.

Beispiel: `O 5000 8000 00` füllt den Bereich von $5000 bis $7FFF mit Nullen.

Man kann mit »OCCUPY« aber nicht nur Speicherbereiche löschen, sondern auch mit beliebigen Werten belegen. Häufig hat man das Problem, festzustellen, welcher Speicherplatz von einem Programm wirklich benutzt wird. Wir füllen den infrage kommenden Bereich dann zuerst zum Beispiel mit »AA« und laden dann unser Programm.

Probieren Sie bitte das folgende Beispiel: Füllen Sie den Speicherbereich von $3000 bis $6000 mit $AA und laden Sie dann unser SUPERTEST-Programm. Beim Disassemblieren können Sie erkennen, dass unser kleines Programm exakt zwischen vielen AAs eingebettet ist.

WRITE

`W (ANFADRalt ENDADRalt ANFADRneu)`
verschiebt den Speicherbereich von ANFADRalt bis ENDADRalt nach ANF-ADRneu ohne Umrechnung der Adressen!

Unser kleines Testprogramm möge noch einmal als Beispiel dienen: `W 4000 4020 6000` verschiebt das oben angesprochene Programm von $4000 nach $6000.

Hierbei werden weder die absoluten Adressen umgerechnet noch die Tabellen geändert. Letzteres ist sicherlich erwünscht, aber denken Sie daran, dass das verschobene Programm nun nicht mehr lauffähig ist, da die absoluten Adressen nicht mehr stimmen (zum Beispiel bei dem Befehl `LDA 400E,Y`). Falls Sie jetzt »`G 6000`« eingeben, um das Programm zu starten, werden Sie sich sicherlich wundern, dass es dennoch läuft.

Doch löschen Sie einmal das Programm in $4000 (mit »`O 4000 4100 AA`«) und starten das Programm in $6000 noch einmal! Seltsam, nicht? Abhilfe schafft der nächste Befehl.

VARIATION

`V (ANFADRalt ENDADRalt ANFADRneu ANFADR ENDADR)`
rechnet alle absoluten Adressen im Bereich von ANFADR bis ENDADR, die sich auf ANFADRalt bis ENDADRalt beziehen, auf ANFADRneu um.

Kompliziert? Nicht, wenn Sie sich klarmachen, dass die ersten drei Adressen exakt den Eingaben beim »`W`«-Befehl entsprechen. Neu hinzukommen nur die beiden Adressen für den Bereich, in dem die Änderung tatsächlich erfolgt.

Um unser mit »`W`« schon verschobenes Programm auch wieder lauffähig zu machen, geben Sie Folgendes ein:

`V 4000 4020 6000 6000 600E`. Damit werden alle Absolutadressen, die im Bereich von $6000 bis $600E - dahinter steht die Tabelle - liegen und sich bisher auf $4000 bis $4020 bezogen haben, auf den neuen Bereich umgerechnet. Probieren geht wie immer über kapieren.

Eine Zusammenfassung dieser beiden Befehle ermöglicht:

CONVERTIEREN (Verschieben eines Programms mit Adressumrechnung)

`C (ANFADRalt ENDADRalt ANFADRneu ANFADRges ENDADRges)`
verschiebt das Programm von ANFADRalt bis ENDADRalt zur ANFADRneu, und zwar mit Umrechnung der Adressen zwischen ANFADRges und ENDADRges.

An unserem kleinen Testprogramm lässt sich wieder einmal demonstrieren, wie der Befehl eingesetzt wird. Laden Sie es also mit »`L "SUPERTEST"`« und schauen es mit »`D 4000`« an.

Jetzt wollen wir an der Adresse $4008 einen 3-Byte-Befehl einfügen: `C 4008 4020 400B 4000 4011` verschiebt das Programm von $4008 bis $4020 zur neuen Anfangsadresse $400B. Dabei werden im Bereich von $4000 bis $4011 (neue Endadresse des »aktiven« Programms!) die Sprungadressen umgerechnet. Nun können Sie ab Adresse

$4008 einen 3-Byte-Befehl einfügen, zum Beispiel STY 0286. Dazu geben Sie bitte ein:

```
A 4008
4008 STY 0286
F
```

Überzeugen Sie sich davon, dass SMON die Befehle korrekt umgerechnet hat, indem Sie unser Beispiel disassemblieren (D 4000) und anschließend mit G 4000 starten.

Besitzer eines Farbmonitors werden in helle Begeisterung ausbrechen. Vorsicht ist geboten, wenn Tabellen oder Text vorhanden sind. SMON wird versuchen, diese als Befehle zu disassemblieren und gegebenenfalls umzurechnen. Dabei können unvorhersehbare Verfälschungen auftreten.

Aus diesem Grunde ist im Beispiel die Endadresse des zu ändernden Bereiches auf $4011 und nicht etwa auf $4023 gelegt worden.

Wenn Sie größere Programme zu verschieben haben, sollten Sie die Kommandos W und V anwenden beziehungsweise einen Assembler einsetzen, der es Ihnen gestattet, beliebige Einfügungen, Verschiebungen und sonstige Änderungen vorzunehmen. Das C-Kommando eignet sich in erster Linie für kleinere Änderungen innerhalb eines Programms.

Der Blick hinter die Kulissen

Wie beim letzten Mal wollen wir noch einen kleinen Blick auf das Programm werfen. Wir haben zwei häufig vorkommende Programmteile ausgewählt. Wenn Sie nach erfolgreichem Eintippen der neuen DATAs einmal mit »D C84E« die »LOAD-SAVE«-Routine listen, sehen Sie, dass diese sehr wenig Platz beansprucht. Das Geheimnis dieser Beschränkung liegt in der Tatsache begründet, dass wir hier auf Betriebssystem-Routinen zurückgegriffen haben. Doch dazu später mehr; erst einmal die angesprochene Routine von Anfang an:

```
LOADSAVE    LDY    #$02
            STY    *$BC
            DEY
            STY    *$B9
            STY    *$BB
            DEY
            STY    *$B7
```

Die Speicherstellen $BB / $BC enthalten jetzt die Adresse $0201, also den Beginn des BASIC-Eingabepuffers. In $B9 befindet sich der Wert 01. Das bedeutet, dass die Sekundäradresse für absolutes Laden voreingestellt ist. Die Speicherstelle $B7 enthält die Länge des Dateinamens, hier erst einmal 0.

```
JSR    GETCHRERR
CMP    #'"'
BNE    LSERROR
```

Überprüft, ob Anführungsstriche eingegeben wurden. Falls nicht, springt unser Programm in die Routine »LSERROR« und bricht ab.

```
LSI    JSR    GETCHRERR
       STA    ($BB),Y
       INY
       INC    *$B7
       CMP    #'"'
       BNE    LSI
```

In diesem Programmteil wird der Filename eingelesen und in die Adresse gespeichert, die in $BB / $BC enthalten ist ($0201).

Gleichzeitig wird $B7 als Zähler für die Namenlänge so lange erhöht, bis das zweite Anführungszeichen auftaucht.

Damit ist der Filename gespeichert, beginnend bei $0201.

```
DEC    *$B7
LDA    I0.NR
STA    *$BA
LDA    *COMMAND
CMP    #'S'
BEQ    SAVE
```

Da die Namenlänge um eins zu groß geraten ist (das letzte Zeichen war ein »"«), muss sie dekrementiert werden.

Die gewählte oder voreingestellte I/O-Nummer (Device-Nummer) soll in $BA gespeichert werden, damit die Betriebssystemroutine nachher das richtige Gerät anspricht. Zum Abschluss überprüft der Compare-Befehl, ob das Kommando »S« gewählt ist, um dann dorthin zu verzweigen.

```
LOAD    JSR    GETRET
        BEQ    LOAD1
        LDX    #$C3
        JSR    GETADR
        LDA    #$00
        STA    *$B9
```

Wir sind nun an der Stelle des Befehls angelangt, an der sich herausstellen muss, ob das Programm an seine Originaladresse (absolut) oder an eine andere Adresse geladen werden soll.

Die Unterroutine »GETRET« prüft, ob unmittelbar nach dem Namen ein <RETURN> folgt und führt eine Verzweigung nach »LOAD1« durch, falls dieses eintritt.

Ansonsten holen wir uns die Adresse und laden das vorgesehene Programm dorthin, nachdem in Speicherstelle *$B9 eine Null gespeichert ist, da ein absolutes Laden nicht erfolgt. Die Routine »GETADR« ist so aufgebaut, dass sie die nächsten zwei Bytes an die mit dem X-Register gewählte Stelle in der Zeropage ablegt. Dann führen wir ebenfalls »LOAD1« durch.

```
LOAD1   LDA    #$00
        JMP    ($0330)
SAVE    LDX    #$C1
        JSR    GETADR
        LDX    #$AE
        JSR    GETADR
        JMP    ($0332)
```

In »LOAD1« erfolgt der indirekte Sprung über $0330 in die LOAD-Routine des Betriebssystems. Die SAVE-Routine erfragt vorher noch die fehlenden Adressen (Anfangs- und Endadresse des Programms, das gespeichert werden soll), speichert sie nach $C1/C2 und $AE/AF und springt dann in die SAVE-Routine.

Noch ein Wort zu den angesprochenen Betriebssystem-Routinen: Mittlerweile gibt es für den C64 mindestens drei verschiedene Versionen des Betriebssystems von Commodore.

Es sind zwar meist nur kleine Änderungen, aber die können fatale Folgen haben, wenn sich die Einsprungadressen ändern. Deshalb gibt es einen besonderen Bereich, das KERNAL, das einen Sprungverteiler für die wichtigsten Routinen enthält. Dieser wird grundsätzlich nie geändert. Beziehen Sie deshalb Ihre Einsprungadressen immer auf die KERNAL-Routinen, um sicher zu gehen, dass Ihr Programm auch noch mit der zwölften Version des Betriebssystems läuft. Die KERNAL-Einsprünge stehen ganz hinten ab $FF81 im Speicher.

Als Zweites ein Vergleich, der in Maschinenprogrammen häufig und in allen Variationen auftaucht: Es handelt sich dabei um den Vergleich zweier Adressen.

Nun sind Adressen leider 16-Bit-Werte, unser Prozessor aber kann nur 8 Bit auf einmal verarbeiten. Gehen wir einmal von folgenden Bedingungen aus: Ein Programm soll von $4000 bis $4020 gelistet werden. Die Zeiger für das Ende befinden sich in Speicherstelle ENDLO (Lowbyte) und ENDHI (Highbyte). »PCL« (Program-Counter-Low) und »PCH« (Program-Counter-High) geben den augenblicklichen Stand des Programms an. Dann erfolgt die Abfrage auf erreichtes Ende mit dieser Befehlsfolge:

```
VERGL   LDA    PCL
        CMP    ENDLO
        LDA    PCH
        SBC    ENDHI
        BCS    FERTIG
WEITER  JMP    AUSGABE
FERTIG  IMP    ENDE
```

Solange PCL und PCH kleiner sind als die Endwerte, geht das Programm »WEITER«.

Sobald aber PCL und PCH die Werte von ENDLO und ENDHI erreicht haben, wird das Carry-Flag gesetzt, und die Abfage mit BCS FERTIG würde das Auflisten anhalten.

Dass es bei der Anwendung einige Probleme geben kann, sieht man daran, dass die Ausgabe auch schon unterbrochen wird, wenn gerade erst das Programmende erreicht ist. (Der letzte Befehl könnte »unter den Tisch fallen«.)

Aber kein Problem ohne Problemlösung - und natürlich ohne weitere Probleme, die Sie aber mit ein bisschen Nachdenken sicher selbst lösen können.

Ohne gutes Werkzeug geht es nicht: SMON Folge 3

Erschienen in: 64'er 1/1985
Verantwortlicher Redakteur: Georg Klinge
Autoren: Dietrich Weineck und Norfried Mann

Der Maschinensprache-Monitor geht langsam seiner Vollendung entgegen. In diesem Teil kommen drei interessante Befehle hinzu, die vor allem bei der Fehlersuche sehr hilfreich sind.

Sicherlich haben Sie sich beim letzten Mal gewundert, wie es möglich ist, dass so viele neue Befehle in so wenig Programm stecken können. »Schuld« daran ist das SMON-Konzept. Wir haben im ersten Teil bereits alle Ein- und Ausgaberoutinen untergebracht. Sämtliche Erweiterungen können nun darauf aufbauen und werden dementsprechend kürzer.

Wir haben Ihnen sogar noch einen Befehl verschwiegen, der beim letzten Mal schon vorhanden war. Vielleicht haben Sie versehentlich einmal »B« eingegeben, und SMON hat mit einem Fragezeichen reagiert und damit gezeigt, dass er mit »B« etwas anzufangen weiß.

Im heutigen Teil 3 unserer SMON-Serie wollen wir Ihnen drei weitere Befehle vorstellen: BASIC-DATA, KONTROLLE und FIND. Es sind diesmal nur drei neue Befehle - nicht, weil wir Sie für den nächsten Artikel über »SMON« auf die Folter spannen wollen, sondern weil wir der Meinung sind, dass ein so umfassender Befehl wie »FIND« schon eine Menge an Beispielen braucht, um verstanden zu werden.

BASIC-DATA

B (ANFADR ENDADR)

wandelt das Maschinenprogramm von ANFADR bis ENDADR-1 in BASIC-DATA-Zeilen um:

```
B 4000 4020
```

Unser Testprogramm (Sie erinnern sich doch noch an unser kleines Programm aus 11/1984?) wird in DATA-Werte umgerechnet und dann mit Zeilennummer 32000 beginnend im BASIC-Speicher abgelegt.

Ein im Speicher befindliches BASIC-Programm (zum Beispiel ein BASIC-Lader) mit kleineren Zeilennummern kann dann diese DATA-Zeilen benutzen.

Wenn Sie das Testprogramm wie oben beschrieben umgewandelt haben, verlassen Sie nun mit »X« den SMON und überzeugen sich mit »LIST« von der Ausführung. Dann können Sie Folgendes eingeben:

```
10 FOR I = 16384 TO 16415 : READ D : POKE I,D : NEXT I
```

In Verbindung mit den oben erzeugten DATA-Zeilen (und »RUN«!) hätten Sie wieder das ursprüngliche Maschinenprogramm im Speicher. Falls Sie dieses Beispiel durchführen wollen, denken Sie bitte daran, dass Sie nach Erstellung der DATAs das Originalprogramm zum Beispiel mit OCCUPY (O 4000 4020 AA) überschreiben, damit Sie die richtige Ausführung überprüfen können.

Der BRK-Befehl am Ende des Testprogramms bewirkt einen Sprung zurück zum SMON. Wollen Sie ein Maschinenprogramm von BASIC aus starten und auch wieder dorthin zurückgelangen, muss der letzte Befehl ein RTS sein. Probieren Sie es aus, indem Sie das BASIC-Programm um `20 SYS 16384` erweitern.

KONTROLLE

`K (ANFADR ENDADR)`

listet die ASCII-Zeichen im gewünschten Bereich. Es werden jeweils 32 Zeichen pro Zeile ausgegeben, so dass man sich einen schnellen Überblick über Texte oder Tabellen verschaffen kann. Beispiel:

```
K 4000
```

listet die ersten 32 Zeichen unseres Programms. Die weitere Ausgabe ist genau wie beim Disassemblieren durch Druck auf <SPACE> oder <RETURN> möglich.

Auch hier können Sie wie bei den anderen Bildschirm-Ausgabebefehlen Änderungen durch einfaches Überschreiben vornehmen (natürlich nicht im ROM und nur mit ASCII-Zeichen!).

Als Beispiel wollen wir einmal im BASIC »herumpfuschen«: Das geht natürlich nicht so ohne Weiteres, weil das BASIC im ROM steht und damit nicht verändert werden kann. Tippen Sie bitte Folgendes ein:

```
W A000 C000 A000
```

Auf den ersten Blick eine unsinnige Anweisung; der Speicher soll von A000 bis C000 nach A000 verschoben werden. Dieser Befehl entspricht exakt der BASIC-Schleife

```
FOR I = 40960 TO 49152 : POKE
I,PEEK (I) : NEXT I
```

Nun ist es aber so, dass beim PEEK das ROM gelesen, beim POKE aber ins darunter liegende RAM geschrieben wird. Wir erreichen also, dass das BASIC ins RAM kopiert wird. Jetzt müssen wir dafür sorgen, dass das Betriebssystem sein BASIC aus dem RAM und nicht aus dem ROM holt.

Zuständig dafür ist die Speicherstelle 0001. Geben Sie bitte »`M 0001`« ein und überschreiben Sie die »37« mit »36«.

Es passiert gar nichts. Jetzt tritt unser K-Kommando in Aktion. Geben Sie ein:

```
K A100 A360
```

Was Sie sehen, sind die BASIC-Befehlswörter und -Meldungen. Schalten Sie mit <SHIFT> + <CBM> auf Kleinschrift, dann erkennen Sie, dass der jeweils letzte Buchstabe eines Befehlswortes groß geschrieben ist (Ende-Kennung).

Jetzt ändern Sie durch Überschreiben das »LIST« (A100) in »LUST« und »ERROR« (A360) in »FAELER«. (Bei »FAELER« müssen Sie ein Zeichen vor »ERROR« beginnen, sonst passt es nicht.)

Verlassen Sie jetzt SMON mit »X« und geben Sie danach ein: POKE 1,54

SMON schaltet nämlich beim »X«-Befehl immer auf das BASIC-ROM zurück, daher müssen wir wieder auf unser geändertes BASIC umschalten. Schreiben Sie nun einen BASIC-Dreizeiler und versuchen Sie, diesen zu LISTen. Ergebnis?

Versuchen Sie es jetzt einmal mit »LUST«. Ihrer weiteren Phantasie sind keine Grenzen mehr gesetzt ...

Wie oben angesprochen, stellt SMON eine Reihe verschiedener Suchroutinen zur Verfügung, die im Folgenden an vielen Beispielen beschrieben werden. Alle diese Befehle bestehen aus zwei Zeichen und beginnen mit dem Buchstaben »F«.

FIND

`F (HEX-WERT(e), ANFADR ENDADR)`

sucht nach einzelnen HEX-Werten innerhalb eines bestimmten Bereichs. Das zweite Zeichen (hinter F) ist hier ein Leerzeichen und darf nicht weggelassen werden!

Die Bereichsangabe kann wie bei allen folgenden Befehlen entfallen, dann wird der gesamte Speicher durchsucht.

Beispiel: Wir suchen alle Befehle LDY #01, also die Werte A0 01 im Bereich von $2000 bis $6000. Eingabe: F A0 01, 2000 6000 (die Leerzeichen zwischen den Hex-Bytes dürfen nicht weggelassen werden!). Es erscheinen alle Speicherstellen, die die gesuchten Bytes enthalten, also zum Beispiel 4000.

FA (Adresse, ANFADR ENDADR)
sucht alle Befehle, die eine bestimmte Adresse als Operanden haben (absolut). Die Adresse braucht nicht vollständig angegeben zu werden, es kann das Jokerzeichen »*« benutzt werden.

Erstes Beispiel - Wir suchen alle JSR FFD2-Befehle im Bereich $2000 bis $6000: FAFFD2,2000 6000. Es erscheinen alle Befehle disassembliert, die FFD2 im Operanden enthalten (also auch LDA FFD2 oder STA FFD2,Y).

Zweites Beispiel - Wir suchen alle Befehle, die auf den Grafikbereich ($D000 bis $DFFF) zugreifen: FAD***,2000 6000

Der Joker kann aber auch zum Beispiel zur Suche im Bereich $D000 bis $D0FF dienen: FAD0**,2000 6000

FR (ADR, ANFADR ENDADR)
sucht nach relativen Sprungzielen. Anders als bei absoluten Sprüngen (JMP, JSR) benutzen die Branch-Befehle eine relative Adressierung, also zum Beispiel »Verzweige 10 vor« oder »37 zurück«. Solche Sprünge lassen sich mit dem FA-Kommando nicht finden. Hier wird »FR« eingesetzt.

Beispiel - Gesucht werden alle Branch-Befehle, die die Adresse $4002 anspringen: FR4002,2000 6000. Natürlich können solche Befehle nur höchstens 128 Byte vom Sprungziel entfernt sein. Die Bereichsangabe ist hier also viel zu groß gewählt (SMON stört dies allerdings nicht). Der Einsatz des Jokers ist hier ebenfalls möglich.

FT (ANFADR ENDADR)
sucht Tabellen im angegebenen Bereich. SMON behandelt dabei alles, was sich nicht disassemblieren lässt, als Tabelle.

Beispiel - Wir suchen Tabellen oder Text im Bereich $2000 bis $6000: FT 2000 6000

FZ (Adr, ANFADR ENDADR)
sucht alle Befehle, die Zeropage-Adressen haben.

Erstes Beispiel: FZC5,2000 6000 findet alle Befehle, die C5 adressieren, also zum Beispiel BIT $C5, LDA (C5),Y etc.

Zweites Beispiel: FZF*,2000 6000 findet alle Befehle, die den Bereich zwischen $F0 und $FF adressieren.

Drittes. Beispiel: FZ**,2000 6000 findet sämtliche Befehle mit Zeropage-Adressierung.

FI (Operand, ANFADR ENDADR)
sucht alle Befehle mit unmittelbarer Adressierung (»immediate«).

Beispiel: Gesucht werden Befehle, die zum Beispiel das Y-Register mit 01 laden. FI01, 2000 6000 findet LDY #01 in Adresse $4000.

Sie sehen: SMON bietet eine Fülle von verschiedensten FIND-Routinen, mit denen alles gesucht und auch gefunden (!) werden kann. Zum Üben wollen wir ein großes Preisausschreiben veranstalten, bei dem Sie zumindest an Erfahrung gewinnen können! Hier sind die Aufgaben, die es zu lösen gilt:

1. Wie oft wird von SMON aus in das Betriebssystem gesprungen ($E000 - $FFFF)?
2. Welche Zeropage-Adressen nutzt SMON?
3. Wo wird die Hintergrundfarbe ($D021), wo die Schreibfarbe ($0286) gesetzt?
4. Wo sind im SMON die Tabellen untergebracht?
5. An einigen Stellen stehen Befehle, die die Register unmittelbar mit dem Highbyte des SMON-Speicherbereichs laden (dez. 49152 - 52208). Rechnen Sie die HEX-Werte aus und suchen Sie die Speicherstellen.

6. An zwei Stellen stehen aufeinander folgend fünf Nullen. Wo?

Notieren Sie Ihre Lösung bitte auf einem Zettel und werfen Sie diesen fort. Der Rechtsweg ist ausgeschlossen ...

Der Lösung ein Stück näher...

Auch heute werden Sie nicht entlassen, ohne dass wir Ihnen einige Tipps für eigene Assemblerprogramme mitgeben. Erinnern Sie sich noch an den in der letzten Ausgabe angesprochenen 16-Bit-Vergleich?

Dieser wird im SMON zum Beispiel dazu benutzt, um festzustellen, ob ein Programmteil weiter durchlaufen werden soll, oder ob das Ende erreicht ist. Das prüft SMON bei fast allen Befehlen. Jüngstes Beispiel sind die FIND-Kommandos.

Zur Erinnerung: Wir wollten zwei 16-Bit-Zahlen vergleichen, der Prozessor kann aber nur mit 8-Bit-Zahlen umgehen.

Wir brauchen dazu einen hoch laufenden Zähler (er heißt in unserem Beispiel »Programmzähler« PC und besteht aus Highbyte PCH und Lowbyte PCL) und einen End-Zeiger (ENDHI und ENDLO). Unser Programm dafür sah folgendermaßen aus:

```
LDA PCL
CMP ENDLO
LDA PCH
SBC ENDHI
```

Anschließend haben wir das Carry-Flag überprüft und festgestellt, dass schon bei Übereinstimmung beider Adressen dieses Flag gesetzt war. In unserem Fall würde also ein Beenden des Programmteils mit »BCS ENDE« zu einer »Unterschlagung« des letzten noch auszuführenden Befehls führen.

Um hier einen Weg aus dem Dilemma zu finden, wollen wir uns das Verhalten der Zero- und Carry-Flagge im Status-Register einmal genauer ansehen - und zwar in Abhängigkeit von PC und dem ENDE-Zeiger. Sie sehen im folgenden Listing, dass wir den Programmzähler (PCL / PCH) nach $FB / $FC und den Endezeiger (ENDLO / ENDHI) nach $FD / $FE schreiben.

Dann springen wir die Routine an, die die Überprüfung auf erreichtes Ende im SMON vornimmt (CMPEND in $C466). Zum Abschluss sorgt der BRK-Befehl dafür, dass wir wieder im SMON landen. Schauen Sie sich die entsprechende Routine im SMON mit »D C466« an. Sie werden erkennen, dass sie den oben angesprochenen 16-Bit-Vergleich durchführt.

Speichern Sie jetzt dieses Programm mit »S "CMP-TEST" 4100 4112« ab und starten es mit »G4100«.

Nachdem das Programm gelaufen ist, meldet es sich mit der Registeranzeige zurück. Achten Sie dabei vor allem auf die Statusregister-Anzeige rechts, denn uns interessieren die Werte für Z und C (Zero-und Carry-Flagge). Tippen Sie bitte nun das folgende Programm ein (mit »A 4100«):

```
4100 LDA #00
4102 STA FB
4104 STA FD    (= PCL)
4106 LDA #C0
4108 STA FC    (= PCH)
410A LDA #C1
410C STA FE    (= ENDHI)
410E JSR C466  (= CMPEND)
4111 BRK
```

Wir wollen herausfinden, was passiert, wenn der Programmzähler (PC) kleiner, gleich oder größer als ENDE ist. Wenn Sie jetzt in Speicherstelle $4106 für PCH den Wert C1 einsetzen, können Sie den Vorgang wiederholen und die Änderung der Flaggen notieren.

Anschließend setzen Sie C2 für PCH in $4106 ein.

Tippen Sie »D 4100 4112«, gehen mit dem Cursor in die Zeile 4106 und überschreiben den Wert #C0 mit dem neuen Wert #C1 beziehungsweise #C2.

		PC<END	PC=END	PC>END
EPC	FC/FB	C0/00	C1/00	C2/00
EEND	FE/FD	C1/00	C1/00	C1/00
		ZC	ZC	ZC
		00	11	01

So sollte Ihre Tabelle zum Schluss aussehen. Im ersten Fall (PC ist kleiner als END) ist das Carry-Flag gelöscht.

Dann (PC ist gleich END) sind Z- und C-Flagge gesetzt, zum Schluss ist nur noch das C-Flag 1.

Jetzt können wir unseren Vorstellungen entsprechend reagieren und mit den Branch-Befehlen verzweigen.

Sehen Sie sich mit dem Disassembler auch einmal andere Routinen im SMON daraufhin an.

Hinweis zum 16-Bit-Vergleich

Im Text steht bei der Beschreibung des 16-Bit-Vergleichs, das Zero-Flag sei gesetzt, wenn beide Werte gleich seien. Das stimmt auch, aber Sie wissen ja selbst, wo der Teufel steckt: nämlich im Detail.

Der Text verführt uns nämlich zu dem eigentlich logischen Schluss, man könne auf die beschriebene Art und Weise prüfen, ob zwei 16-Bit-Werte gleich sind. Und genau das geht nicht!

Das Zero-Flag hat nämlich die unangenehme Eigenschaft, auch dann gesetzt zu sein, wenn der erste Wert bis zu 255 größer (!) ist als der zweite. Will man also zuverlässig auf Gleichheit testen, muss man die beiden Werte voneinander abziehen und nachsehen, ob das Ergebnis Null ist.

Ohne gutes Werkzeug geht es nicht: SMON Folge 4

Erschienen in: 64'er 2/1985
Verantwortlicher Redakteur: Georg Klinge
Autoren: Dietrich Weineck und Norfried Mann

Der Trace-Modus, also das Abarbeiten von Maschinenprogrammen Schritt für Schritt, ist das wohl wichtigste Hilfsmittel beim Austesten von Maschinenprogrammen.

SMON bietet sogar drei verschiedene Trace-Möglichkeiten, die in diesem vierten Teil dargestellt werden. Außerdem erhalten Sie eine Übersicht aller Befehle, und wir zeigen Ihnen, wie Sie SMON in einen anderen Bereich verschieben können.

Einen Befehl haben wir Ihnen bisher unterschlagen, der zwar bereits vorhanden war, aber noch nicht beschrieben wurde. Es handelt sich um den Vergleich zweier Speicherbereiche. Die Syntax ist sehr einfach:

```
= 4000 6000
```

vergleicht den Speicherinhalt ab $4000 mit dem ab $6000. Das erste nicht übereinstimmende Byte wird angezeigt, und der Vergleich wird abgebrochen.

Wenn Sie also ein Maschinenprogramm geschrieben und überarbeitet haben und Sie wissen nicht mehr genau, worin eigentlich der Unterschied zwischen der 76. und der 77. Version besteht, gehen Sie so vor:

Laden Sie zuerst Version 76 und verschieben Sie diese mit dem »W«-Befehl in einen freien Speicherbereich. Laden Sie dann Version 77 und führen Sie den »=«-Befehl durch. Sofort finden Sie den Unterschied und können mit der Arbeit an Version 78 beginnen ... Wir wollen uns bei der Beschreibung der Trace-Befehle auf Anwendungsbeispiele konzentrieren. Zum Aufbau der Routine sei nur soviel gesagt: Gesteuert wird sie mit Hilfe des Prozessor-Interrupts, weil nur damit ein Eingriff ins laufende Maschinenprogramm möglich ist.

Während des Trace-Ablaufs wird deswegen der Bildschirm kurzfristig aus- und eingeschaltet, weil alle anderen Interruptanforderungen wie beispielsweise durch den Video-Chip, verhindert werden müssen.

Da die Befehle eines Programms nicht nur angezeigt, sondern auch wirklich ausgeführt werden, ist der »SEI«-Befehl mit großer Vorsicht zu verwenden. Doch dazu später mehr. Wir wollen ein neues, besser geeignetes Beispiel verwenden als bisher. Tippen Sie also das folgende Miniprogramm mit dem Assembler ein (A 4000):

```
4000 LDA #30    lade den Akku mit (ASCII-) 0
4002 JSR FFD2   gib Akku auf dem Bildschirm aus
4005 CIC
4006 ADC #01    erhöhe Akku um 1
4008 CMP #39    vergleiche Akku mit (ASCII-) 9
400A BCC 4002   springe zurück, wenn Akku kleiner
400C BRK        springe in SMON zurück
```

Starten Sie das Programm mit »G 4000«. Es muss die Zahlen von 0 bis 8 auf den Bildschirm schreiben.

Trace-Stopp

TS (Startadresse Stoppadresse)

Starten Sie nun unser Programm mit TS

4000 4009. Die ersten Befehle werden ausgeführt (die Null ausgegeben, der Akku erhöht etc.), dann stoppt das Programm bei Adresse $4009 und springt in die Registeranzeige.

Genau genommen ist »TS« gar kein Trace-Befehl, das Programm läuft nämlich bis zur gewählten Stoppadresse in Echtzeit durch. Dort angekommen, können Sie die Register prüfen und gegebenfalls durch Überschreiben ändern. Mit »G«, »TW« oder »TB« (wird später erklärt) ohne weitere Adresseneingaben können Sie dann im Programmlauf fortfahren. SMON merkt sich nämlich, wo er stehen geblieben ist, und arbeitet ab dieser Adresse weiter, wenn Sie nicht eine neue angeben.

Sinnvoll ist dieser Befehl immer dann, wenn in einem längeren Programm nur bestimmte Teile »getraced« werden sollen, der Anfang aber durchlaufen werden muss, um Variablen zu setzen oder Benutzereingaben zu erfragen. Auch wenn man nicht ganz sicher ist, ob eine bestimmte Passage überhaupt jemals durchlaufen wird, kann man das mit »TS« überprüfen. Zwei Einschränkungen gibt es allerdings wegen der Arbeitsweise dieses Befehls: SMON setzt im Programm an die Stoppadresse einen BRK-Befehl und merkt sich, welcher Befehl dort stand, um ihn wieder zurückzuschreiben.

Deshalb funktioniert »TS« nur im RAM, nicht aber zum Beispiel im BASIC oder im Betriebssystem. Auch darf die Speicherstelle, in der sich SMON den ausgetauschten Befehl merkt ($02BC), vom Programm nicht verändert werden, sonst ist eine korrekte Reparatur nicht mehr möglich.

Der wohl am häufigsten und vielseitigsten eingesetzte Trace-Befehl ist sicherlich »TW«.

Trace Walk

TW (Startadresse)

Starten Sie unser Beispiel jetzt mit TW 4000. Der erste Befehl (LDA #30 in Adresse $4000) wird ausgeführt, SMON stoppt und zeigt dann die Inhalte aller Register in der gleichen Reihenfolge wie beim »R«-Kommando sowie den nächsten Befehl an. Im Akku steht jetzt 30, der Programmzähler zeigt auf $4002. Jetzt drücken Sie eine Taste. Der nächste Befehl (JSR FFD2) wird ausgeführt, der Programmzähler zeigt auf $FFD2.

Achten Sie auf den Stack Pointer: Sein Inhalt hat sich um 2 vermindert, weil der Prozessor auf dem Stack die Adresse abgelegt hat, an die er nach Beendigung der Subroutine zurückspringen soll. Der nächste angezeigte Befehl ist ein indirekter Sprung über $0326. Mit dem nächsten Tastendruck wird er durchgeführt.

Und so geht es munter weiter. Verzweifeln Sie nicht, wenn Sie auch nach den nächsten zehn Tastendrücken immer noch irgendwo im Betriebssystem »herumtracen« und von unserem Beispielprogramm weit und breit nichts mehr zu sehen ist.

Ausnahmsweise ist unser Liebling einmal nicht im »Land der Träume« verschwunden, sondern tut, was er soll: Er arbeitet brav einen Befehl nach dem anderen alles ab, was zur Routine $FFD2 gehört, und das ist reichlich viel. Also bewegen Sie Ihre Finger, Sie haben's ja nicht anders gewollt!

Irgendwann einmal, nach mehreren hundert gedrückten Tasten, befinden Sie sich plötzlich wieder in der Registeranzeige von SMON. Das Programm ist beendet. Nun werden Sie enttäuscht fragen, was man wohl mit einem Trace-Modus anfangen soll, der schon bei kleinsten Beispielprogrammen ein völlig undurchschaubares Chaos erzeugt? Nur Geduld, die Rettung naht in Gestalt der Taste <J>.

Falls ihre Hand noch nicht in Gips liegt, starten Sie das Ganze nochmal von vorn mit »TW 4000«. Diesmal drücken Sie aber jedes Mal, wenn als nächster Befehl »JSR FFD2« angezeigt wird, auf <J>.

Der Effekt ist, dass die gesamte Subroutine auf einen Schlag abgearbeitet wird und Sie sofort wieder auf dem nächsten Befehl unseres Beispiels landen. Dass wir nicht gemogelt und die Befehle von »JSR FFD2«

einfach unterschlagen haben, sehen Sie daran, dass der Akku tatsächlich auf dem Bildschirm ausgegeben worden ist (rechts neben FFD2).

Jetzt können Sie unser Beispiel in aller Ruhe bis zum Ende durchgehen und verfolgen, wie der Akku erhöht wird, wie der Vergleich das Statusregister beeinflusst und wie dementsprechend der Rücksprung in die Schleife erfolgt.

Sie dürfen die <J>-Taste auch dann benutzen, wenn Sie schon mitten in der Subroutine sind. Aber hierbei ist äußerste Vorsicht geboten: Die Rücksprungadresse muss unbedingt oben auf dem Stack liegen, wenn Sie <J> drücken. Hat nämlich der Prozessor Werte auf dem Stack abgelegt (mit PHA oder PHP), dann erfolgt der Sprung irgendwohin, nur nicht zurück ins Programm.

Achten Sie deshalb genau auf die Anzeige des Stack Pointers. Wenn dessen Wert genauso groß ist wie bei Beginn der Subroutine, kann nichts passieren. Sonst hilft nur noch der Reset-Taster, den Sie ja inzwischen hoffentlich eingebaut haben, oder eine ruhige Hand, die die Büroklammer an Pin 1 und 3 des User-Ports hält (Kostenpunkt der Reparatur bei Abrutschen: zirka 100 Mark ...)

»TW« bricht automatisch mit der Registeranzeige ab, wenn im Programm ein »BRK«-Befehl auftaucht. Wenn Ihnen das zu lange dauert oder Sie zwischendurch ein Register ändern möchten, können Sie den Trace-Modus jederzeit mit der Stopp-Taste verlassen. Anschließend können Sie wie bei »TS« beschrieben fortfahren.

Im Gegensatz zu »TS« können Sie mit »TW« auch im ROM herumstöbern; Sie haben es ja bei der Subroutine $FFD2 bereits getan. Einzige Einschränkung beim »TW«-Befehl: Ihr Programm darf keinen »SEI« enthalten, da dieser den Interrupt und damit auch den Trace-Modus lahm legt.

Verlassen Sie in diesem Falle »TW« mit STOP und starten erneut hinter dem »SEI«-Befehl. Allerdings müssen Sie in Kauf nehmen, dass das Programm normalerweise nicht mehr korrekt arbeitet. Das nächste Programm soll als weiteres Beispiel für den TW-Modus dienen. Geben Sie es folgendermaßen ein:

```
5000 LDA #00    lädt den Akku mit »0«
5002 TAX        überträgt den Akku ins X-Register
5003 .0C        ein mysteriöses Byte
5004 LDA #04    lädt den Akku mit »4«
5006 TAY        überträgt den Akku ins Y-Register
5007 BRK        springt in SMON
```

Wenn wir dieses kleine Programm abarbeiten, müsste das X-Register auf »0« stehen, während Akku und Y-Register mit »4« geladen sind. Starten wir also das Programm mit »G 5000« und schauen uns die Register an.

Seltsamerweise enthalten alle Register eine »0«. Vorsichtig, wie wir sind, überschreiben wir die drei Register mit »FF«, um die Veränderung deutlich kontrollieren zu können.

Dann starten wir mit »G 5000« ein zweites Mal. Gegen alle Gesetze der Vernunft erscheint wieder das »falsche« Ergebnis - alle drei Register sind »0«. Hier soll uns jetzt der TW-Modus weiterhelfen, indem er uns zeigt, was in Wirklichkeit passiert.

Geben wir »TW 5000« ein. Der erste Befehl (LDA #00) ist durchgeführt, im Akku erscheint die Null. Jetzt steht der Programmzähler auf dem folgenden Befehl »5002 TAX«. Nach Drücken einer Taste wird dieser Befehl ausgeführt, und es erscheint die Null im X-Register. Beim folgenden Befehl müssen wir feststellen, dass der Disassembler nicht in der Lage ist, ihn zu interpretieren - er gibt drei Sternchen aus. Hierbei handelt es sich um unser Byte »0C«.

Wieder ein Tastendruck; und dann erkennen wir, dass etwas Merkwürdiges passiert ist. Der Prozessor hat augenscheinlich den nächsten Befehl (LDA #04) übersprungen und steht schon auf dem folgenden »TAY«. So also wird unser Programm abgearbeitet. Damit ist auch das »falsche« Ergebnis erklärt. Bleibt nur noch die Frage nach dem Grund

für dieses seltsame Verhalten. Und der ist sicherlich in dem mysteriösen Byte »0C« zu suchen. Hierbei handelt es sich um einen der »inoffiziellen« Opkodes (auch Pseudo-Opkodes genannt), die auf Grund der Prozessorarchitektur vorhanden sind und in manchen Programmen ihr Unwesen treiben - wie wir zu unserem Leidwesen erfahren mussten. Das Byte »0C« wirkt wie ein »NOP«, der eine Länge von 3 Byte hat. Deshalb wird der folgende 2-Byte-Befehl (LDA #04) verschluckt. Es gibt noch Einiges zu entdecken am 6502 und 6510 - TW macht's möglich.

Häufig ist es nicht sinnvoll, ein Programm von Anfang an im TW-Modus laufen zu lassen. Zum Anderen sind gerade Schleifen, die per Hand mit »TW« durchlaufen werden müssen, eine ermüdende Angelegenheit. Hier bietet SMON neben dem bereits beschriebenen »TS« eine weitere Trace-Möglichkeit an

Trace Break
TB (Adresse Anzahl der Durchläufe)

Trace Quick
TQ (Adresse)

Geben Sie als Beispiel folgendes Programm ein:

```
6000 LDY #00       Y als Zähler auf »0«
6002 LDA 600E,Y    Werte von $600E ff. sollen
                   geladen werden
6005 JSR FFD2      Ausgabe der Zeichen auf dem
                   Bildschirm
6008 INY           der Zähler wird erhöht
6009 CPY #0E       Zähler schon »14«?
600B BCC 6000      wenn nein, dann nächsten
                   Wert holen
601D BRK
```

Bei $600E soll nun ein Text stehen, den das Programm ausgibt. Die einfachste Art, mit SMON Texte in den Speicher zu schreiben, besteht im »K«-Befehl. Geben Sie K 600E ein (danach natürlich <RETURN>) und drücken Sie die <STOP>-Taste. Fahren Sie mit dem Cursor an das erste ausgegebene Zeichen (vermutlich ein Punkt) und schreiben Sie - ohne Anführungszeichen -: »FEHLER BEHOBEN«.

Drücken Sie dann <RETURN>, um die Zeile an den Rechner zu übergeben. Wenn Sie das Programm starten, werden Sie wieder einmal Gelegenheit haben, sich in Ruhe etwas zu trinken zu holen (Prost!), denn das Programm enthält einen dummen Fehler und beschäftigt den Computer für eine lange, lange Zeit. Genauer gesagt, bis Sie ihn mit Reset (zum Beispiel durch <RUN/STOP> + <RESTORE>) erlösen.

Nun soll SMON helfen, diesen Fehler zu lokalisieren. Setzen Sie zuerst einmal einen Breakpoint bei $6002 und begrenzen die Durchläufe auf die maximale Anzahl: TB 6002 0E und starten mit TQ 6000 den Quicktrace bei $6000.

Das Programm läuft so lange, bis zum 14. Mal die Adresse $6002 erreicht wird und springt dann in den TW-Modus. Wenn Sie sich jetzt die Registerinhalte genau anschauen, müsste Ihnen der Fehler geradezu ins Auge springen.

Wie groß sollte denn das Y-Register sein? Welchen Wert sollte der Akku haben? NA?! (Auflösung erfolgt mit der Bekanntgabe der Gewinner des großen Preisausschreibens aus der letzten Folge ...)

Wenn Sie eine Zeit lang mit SMON gearbeitet haben, werden Ihnen eventuell einige Voreinstellungen nicht gefallen. Besitzer einer Datasette zum Beispiel müssen jedesmal mit »I 01« auf ihre Geräteadresse einstellen.

Wenn Sie sich an unserem »Preisausschreiben« vom letzten Mal beteiligt haben, dürften Ihnen diese Speicherstellen bereits bekannt sein.

Übrigens sind zirka 235 982 richtige Lösungen eingegangen; wir haben daraufhin ein Programm geschrieben, um den Gewinner zu ermitteln. Dieses läuft zur Zeit auf einem Großrechner in den USA, da der Spei-

cher des C 64 nicht ausreichte. Wir werden Sie nach Abschluss des Programmlaufs, den wir für Mitte Juni diesen Jahres erwarten, informieren ...

Sollten Sie nicht zu den glücklichen Gewinnern zählen, geben wir Ihnen im Folgenden die Speicherstellen an, in denen die Voreinstellungen stehen. Diese können Sie dann mit dem »M«-Befehl Ihren Wünschen entsprechend abändern.

Vergessen Sie aber nicht, Ihre geänderte Version mit »S "@:SMON $C000" C000 CE09« abzuspeichern.

Hintergrundfarbe	$C220
Schreibfarbe	$C228
Geräte-Adresse Drucker	$C21B
Geräte-Adresse F1 / Kassette	$C216

Wer steht wo? – Wegweiser durch SMON

Wollen Sie Routinen in SMON analysieren oder verändern, müssen Sie die Einsprungadressen der einzelnen Befehle kennen. Diese lassen sich leicht finden:

Am Anfang des Programms (ab $C00B) befindet sich eine Liste aller Kommandos gefolgt von den Adressen (ab $C02B), bei denen diese Befehle beginnen. Lassen Sie sich mit »M C00B C02B« die Befehlsliste anzeigen (siehe Befehlsübersicht). Sie sehen, dass am Ende 5 Nullen stehen; hier können Sie eigene neue Befehle einbauen.

Mit »M C02B C06B« erhalten Sie die Liste der Einsprungadressen, immer in der Reihenfolge Low-Byte - High-Byte, diesmal mit 10 Nullen am Ende, weil ja zu jedem Befehl 2 Byte für die Adresse gehören.

Wir wollen nun als Beispiel heraussuchen, wo die Routine zum Füllen eines Speicherbereichs mit einem vorgegebenen Byte steht. Der Befehl dazu ist »O«, also der 20. Befehl ($C01E). Die zugehörige Adresse ist ebenfalls die 20. in der Liste, also $C9C0 (in Speicherstelle $C051 / $C052).

Die Routine beginnt allerdings immer erst ein Byte später, in unserem Fall also bei $C9C1.

Disassemblieren Sie jetzt diese Routine mit »C C9C1 C9D3«. Sie erhalten folgende Befehle (natürlich ohne Kommentare):

```
      JSR C27A      holt zwei Adressen nach
                    $FB/FC und $FD/FE
      JSR C28D      holt das gewünschte Byte in
                    den Akku
      LDX #00       initialisiert X-Register als Zähler
LOOP  STA (FB,X)    speichert den Akku-Inhalt in
                    der ersten Adresse ab
      PHA           merkt sich den Akku-Inhalt
      JSR C463      erhöht die Adresse und
                    vergleicht mit der Endadresse
      PLA           holt Akku-Inhalt zurück
      BCC C9C9      wenn Ende nicht erreicht,
                    dann Sprung auf LOOP
      RTS
```

Wenn Sie nun zum Beispiel den »O«-Befehl nur zum Löschen verwenden möchten, könnten Sie die Routine so abändern, dass nur die beiden Adressen eingegeben werden müssen. Überschreiben Sie einfach den Befehl JSR C28D mit LDA #00 im Disassemblerlisting, disassemblieren Sie erneut mit »D C9C1 C9D3« und überschreiben Sie die drei Sternchen bei C9C6 mit einem »NOP«, um die entstandene Lücke (3-Byte-Befehl wurde durch 2-Byte-Befehl ersetzt) aufzufüllen.

Nun können Sie Ihre veränderte Routine ausprobieren. Geben Sie zum Beispiel »O 5000 5200« ein und überzeugen Sie sich mit »M 5000 5200« von der korrekten Ausführung. Sie können auch die so veränderte Routine mit »W C9C1 C9D3 CE09« ans Programmende kopieren und sich einen neuen Befehl »E« (Erase) schaffen.

Sie brauchen dann nur in die erste Null am Ende der Kommandotabelle »45« (für »E«) und in die ersten beiden Nullen der Adresstabelle »08« und »CE« zu schreiben (für die Adresse $CE09).

Machen Sie sich ruhig einmal die Mühe, auch andere Routinen im SMON auf diese Art und Weise zu analysieren und zu ändern. Erstens macht es Spaß und zweitens

können Sie SMON Ihren eigenen Wünschen anpassen. Vielleicht fallen Ihnen ja noch Routinen ein, die in SMON fehlen. Wir würden uns über Verbesserungsvorschläge freuen.

Das »Gedächtnis« von SMON

Wenn Sie Programme mit SMON untersuchen oder verändern wollen, müssen Sie noch wissen, welche Speicherstellen SMON verwendet. Es soll ja Monitorprogramme geben, die die BASIC-Zeiger als Arbeitsspeicher benutzen, so dass ein BASIC-Programm nach dem Rücksprung aus dem Monitor gelöscht ist.

SMON tut so etwas nicht. Aber natürlich braucht auch er Speicherstellen, um sich Werte merken zu können. Damit Sie Konflikten von Anfang an aus dem Wege gehen können, sind die wichtigsten hier dargestellt.

In der Zeropage belegt SMON den Bereich von $00A4 bis $00B6. Dort stehen Systemvariablen für die Kassettenspeicherung und die RS232-Schnittstelle. Diese werden nur während des Betriebs der Kassette oder von RS232 gebraucht, sind ansonsten aber frei.

Außerdem werden die Speicherstellen $00FB bis $00FF benutzt, die sowieso zur freien Verfügung des Anwenders vorgesehen sind. Alle anderen Zeiger in der Zeropage, also insbesondere die Speicherverwaltung für BASIC bleiben unbeeinflusst.

Als weiteren Arbeitsspeicher benutzt SMON den Bereich von $02A8 bis $02C0. Auch dieser Bereich wird vom Betriebssystem nicht benutzt, so dass keine Konflikte entstehen dürften.

Beim Assemblieren wird zusätzlich noch der Kassettenpuffer als Speicher für die Labels benötigt. Dieser bleibt ansonsten aber auch unverändert; das ist wichtig, wenn Maschinenroutinen dort abgelegt werden sollen.

Alles in allem ist SMON also recht verträglich.

SMON verschieben? Mit SMON!

Eine Reihe von Anfragen hat uns erreicht, ob man SMON nicht mithilfe des »W«-, »V«- oder »C«-Kommandos verschieben könne. Alle Versuche in dieser Richtung seien fehlgeschlagen. Einige Leser meinten auch, in der V-Routine müsse ein Fehler stecken. Diesmal sind wir jedoch völlig schuldlos; es gibt nämlich einige Befehle in SMON, die keine Sprungadressen sind und sich trotzdem auf den Bereich ($C000-) beziehen, in dem SMON steht.

Dazu gehören in erster Linie die oben erwähnten Einsprungadressen, deren High-Byte natürlich geändert werden muss, wenn SMON in einem anderen Speicherbereich laufen soll. Es gibt aber auch Befehle, die eine Adresse im Programm in einem Vektor ablegen müssen. Disassemblieren Sie einmal den Anfang von SMON mit »D C000 C00B«. Sie erhalten

```
LDA #14   Low-Byte der BREAK-Routine von SMON
STA 0316  im Break-Vektor abspeichern
LDA #C2   High-Byte (siehe oben!)
STA 0317  siehe oben
BRK
```

Damit wird der Break-Vektor des Betriebssystems auf den SMON gesetzt und mit dem anschließenden - und jedem weiteren - BRK-Befehl springt das Programm in SMONs BREAK-Routine.

Wenn SMON in einem anderen Bereich als $C000 laufen soll, dann müssen diese Befehle geändert werden.

Heraussuchen kann man sie mit »FIC*, C000 D000«. Sie wissen doch noch, was diese Anweisung bedeutet: Suche mir alle Befehle, die ein Register unmittelbar mit einem Wert laden, der mit $C beginnt.

Aber Vorsicht! Nicht alles, was da angezeigt wird, muss auch geändert werden!

Um Ihnen weitere Stunden sinnlosen Herumbrütens zu ersparen, wollen wir als Beispiel zeigen, wie man SMON in den Bereich $9000 bis $A000 verlegen kann. Natürlich

gilt das im Prinzip für jeden anderen Bereich genauso; wir selbst haben insgesamt fünf SMON-Versionen für fünf verschiedene Speicherbereiche, von denen eine immer passt.

1. Wir verschieben zuerst das ganze Programm ohne Umrechnen in den neuen Bereich: `W C000 CE09 9000`
2. Nun lassen wir alle absoluten (3-Byte-)Befehle umrechnen. Die Tabellen am Anfang von SMON bleiben verschont: `V C000 CE09 9000 9214 9E09`
3. Als Nächstes ändern wir die High-Bytes der Befehlsadressen. Geben Sie `M 902B 906B` ein und ändern Sie in jedem zweiten Byte das »C« durch Überschreiben in »9«. Vergessen Sie nicht, am Ende jeder Zeile <RETURN> zu drücken, damit Ihre Änderung auch übernommen wird.
4. Als Letztes bleiben noch die oben beschriebenen Direktlade-Befehle. Sie müssen so geändert werden, dass sie sich auf den neuen Bereich $9... beziehen. Suchen Sie sie mit `FIC*,9000 9E09` heraus. Sie erhalten

```
9005 LDA #C2     ändern
908B JMP C97D    ändern
9124 CPX #C0     nicht ändern
9386 LDY #C0     ändern
9441 CMP #C0     nicht ändern
987F LDX #C3     nicht ändern
988D LDX #C1     nicht ändern
9992 LDA #C1     nicht ändern
9C2C LDA #CC     ändern
9C5B LDA #C2     ändern
9CF4 LDA #CC     ändern
9DA1 LDX #CC     ändern
9E03 LDA #CC     ändern
```

Ändern Sie nur in den gekennzeichneten Zeilen das »C« in »9« um. Sie sehen, es gibt keine Regel, welche Befehle zu ändern sind und welche nicht. Aus diesem Grund müssen Sie diese Änderungen »von Hand« vornehmen. Speichern Sie jetzt die fertige Version unbedingt mit `S "SMON $9000" 9000 9E09` ab. Nun starten Sie die 9000-Version mit `»G 9000«` oder von BASIC aus mit SYS 36864 (SYS 9 x 4096). Löschen Sie dann den alten SMON mit `»O C000 D000 00«`.

Nur so können Sie sicher sein, dass der verschobene SMON auch einwandfrei arbeitet und Sie nichts übersehen haben.

Probieren Sie nun alle Befehle durch. Sie müssen genauso arbeiten wie bisher. Vor allem können Sie jetzt auch Programme wie »DOS 5.1« oder »Turbo Tape« untersuchen, die im $C000-Bereich stehen. Achten Sie aber, wenn Sie »SMON $9000« von BASIC aus benutzen, darauf, dass das BASIC ihn nicht überschreibt. String-Variablen werden nämlich von $A000 nach unten hin aufgebaut, und bis $9E09 ist nicht viel Platz.

Schützen Sie im Zweifelsfall den Bereich, indem Sie nach dem Laden des SMON $9000 eingeben:

```
NEW : POKE 56,144 : POKE 55,0
```

Damit ist SMON vor Überschreiben geschützt. Das ist natürlich bei dem SMON $C000 nicht nötig, weil BASIC in diesen Bereich nicht hineinkommt.

Wir hoffen, Ihnen mit SMON ein wirklich brauchbares Werkzeug an die Hand gegeben zu haben. Die vielen Zuschriften und Anrufe, die wir im Verlauf dieser Serie erhalten haben, bestätigen uns in dieser Auffassung. Viele Leser sind unserem Eindruck nach dabei, in die Maschinensprache einzusteigen und deren Möglichkeiten zu nutzen. Und genau das wollten wir erreichen.

Damit wären wir eigentlich am Ende dieser Serie angelangt und würden uns mit den üblichen guten Wünschen verabschieden. Aber die freien Bytes am Ende von SMON, immerhin von $CE09 bis $D000, also über 500 Byte, haben uns keine Ruhe gelassen.

Als »Zugabe« werden wir Ihnen deshalb in der nächsten Ausgabe noch einen kleinen Diskettenmonitor vorstellen, der, in SMON eingebunden, dessen Möglichkeit nutzen kann.

SMON-Befehlsübersicht

A 4000 (Assembler)
Symbolischer Assembler (Verarbeitung von Labels möglich). Startadresse $4000

B 4000 4200 (BASIC-DATA)
erzeugt BASIC-DATA-Zeilen aus Maschinenprogramm im Bereich von $4000 - $41FF.

C 4010 4200 4013 4000 4200 (Convert)
In ein Programm, das von $4000 bis $4200 im Speicher steht, soll bei $4010 ein 3-Byte-Befehl eingefügt werden. Dazu wird das Programm ab $4010 - $4200 auf die neue Adresse $4013 verschoben. Alle absoluten Adressen, die innerhalb des Programmbereichs ($4000 - $4200) stehen, werden umgerechnet, so dass die Sprungziele stimmen.

D 4000 (4100) (Disassembler)
disassembliert den Bereich von $4000 (- $4100) mit Ausgabe der Hex-Werte. Änderungen sind durch Überschreiben der Befehle möglich.

F (Find)
findet Zeichenketten (F), absolute Adressen (FA), relative Sprünge (FR), Tabellen (FT), Zeropageadressen (FZ) und Immediate-Befehle (FI).

G (4000) (Go)
startet ein Maschinenprogramm, das bei $4000 im Speicher beginnt.

I 01 (I/O-Gerät)
stellt die Gerätenummer für Floppy (08 oder 09) oder Datasette (01) ein.

K A000 (A500) (Kontrolle)
zum schnellen Durchsuchen des Bereichs von $A000 (- $A500) nach ASCII-Zeichen (32 Byte pro Zeile). Änderungen sind durch Überschreiben der ASCII-Zeichen möglich.

L (4000) (Load)
lädt ein Maschinenprogramm an die richtige oder eine angegebene Adresse ($4000).

M 4000 (4400) (Memory-Dump)
gibt den Inhalt des Speichers von $4000 (- $43FF) in Hex-Bytes und ASCII-Kode aus. Änderungen sind durch Überschreiben der Hex-Zahlen möglich.

O 4000 4500 AA (Occupy)
füllt den Speicherbereich von $4000-$4500 mit vorgegebenem Byte ($AA) aus.

P 02 (Printer)
setzt Geräteadresse 2 für Drucker.

R (Register)
zeigt die Registerinhalte und Flags an. Änderungen sind durch Überschreiben möglich.

S "Test" 4000 5000 (Save)
speichert ein Programm von $4000 - $4FFF unter dem Namen »Test« ab.

TB 4010 05 (Trace Break)
setzt einen Haltepunkt für den Schnellschrittmodus bei $4010. Der Schnellschrittmodus wird unterbrochen, nachdem $4010 zum 5. Mal erreicht worden ist.

TQ 4000 (Trace Quick)
Schnellschrittmodus. Springt beim Erreichen eines Haltepunktes in den Einzelschrittmodus.

TS 4000 4020 (Trace Stop)
arbeitet ein Programm ab $4000 in Echtzeit ab und springt beim Erreichen von $4020 in die Registeranzeige. Von dort aus kann (nach evtl. Änderung der Register) mit »G« oder »TW« fortgefahren werden. »TS« arbeitet nur im RAM-Speicher.

TW (4000) (Trace Walk)
führt auf Tastendruck den jeweils nächsten Maschinenbefehl aus und zeigt die Registerinhalte an. Subroutinen können in Echtzeit durchlaufen werden (<J>). Wird keine Startadresse eingegeben, beginnt TW bei der letzten mit »R« angezeigten Adresse.

V 6000 6200 4000 4100 4200 (Verschieben)
ändert in einem Programm von $4100 - $41FF alle absoluten Adressen, die sich auf den Bereich von $6000 bis $6200 beziehen, auf einen neuen Bereich, der bei $4000 beginnt.

W 4000 4300 5000 (Write)
verschiebt den Speicherinhalt von $4000 - $42FF nach $5000 ohne Umrechnung der Adressen (zum Beispiel Tabellen).

X (Exit)
springt aus dem Monitor-Programm ins BASIC zurück.

 # 49152
Dezimalzahl umrechnen
$ 002B
4-stellige Hex-Zahl umrechnen

% 0110 1010
8-stellige Binärzahl umrechnen

? 0344 + 5234
Addition oder Subtraktion zweier 4-stelliger Hex-Zahlen

= 4000 5000 (Vergleich)
vergleicht den Speicherinhalt ab $4000 mit dem ab $5000.

Hinweise zur Befehlsübersicht
Alle Eingaben erfolgen in der hexadezimalen Schreibweise. In Klammern angegebene Adresseingaben können entfallen.

SMON benutzt dann sinnvolle vorgegebene Werte. Bei allen Ausgabe-Befehlen ist gleichzeitig die Ausgabe auf einen Drucker möglich. Dazu werden die Befehle geSHIFTet eingegeben.

Ohne gutes Werkzeug geht es nicht: SMON Folge 5

Erschienen in: 64'er 2/1985
Verantwortlicher Redakteur: Georg Klinge
Autoren: Dietrich Weineck und Norfried Mann

Dies ist der letzte Teil unserer SMON-Serie mit der versprochenen Zugabe. Die noch freien zirka 500 Byte sollen mit einem kleinen Diskettenmonitor gefüllt werden. Wir wollen Ihnen außerdem an einigen Beispielen zeigen, was man mit so einem Disk-Monitor alles anstellen kann.

Nun sind 500 Byte nicht gerade viel - sie entsprechen etwa 10 Zeilen BASIC -, und das stellte uns vor einige Probleme. Wir haben schließlich auf allen Komfort verzichten müssen: Deshalb gibt es keine Directory-Ausgabe, keine Übermittlung von Disk-Kommandos à la DOS 5.1 und was dergleichen schöne Dinge mehr sind. Niemand kann Sie allerdings davon abhalten, solche Programme gleichzeitig mit SMON zu verwenden.

Sollte SMON dabei im Weg sein, können Sie ihn ja seit der letzten Ausgabe auf einen anderen Speicherbereich verschieben. Wir haben also nur solche Befehle eingebaut, die BASIC-Befehlserweiterungen nicht bieten und die für den Maschinensprache-Programmierer von besonderem Interesse sind.

Glücklicherweise sind gerade für solche Zwecke im SMON bereits reichlich Routinen vorhanden, die wir benutzen können, etwa Eingabe-Routinen oder Hexdump von Speicherbereichen.

Die Befehle des Disk-Monitors

Da das Arbeiten mit dem Disk-Monitor besondere Aufmerksamkeit verlangt (nach Murphys Gesetzen führen Fehleingaben in der Regel zu unlesbaren Disketten), wird er mit einem eigenen Kommando eingeschaltet. Leider waren alle halbwegs sinnvollen Buchstaben (»D« wie Diskette oder »F« wie Floppy) schon vergeben, deshalb haben wir uns für ein schlichtes »Z« wie Zuversicht entschieden.

Z

schaltet den Disk-Monitor ein. Die Rahmenfarbe ändert sich auf Gelb, der gewohnte ».« am Anfang einer Zeile ändert sich in »*«. Dies alles hat den Zweck, Ihnen deutlich zu machen, dass es jetzt ernst wird. Intern wird jetzt das BASIC abgeschaltet, weil der Disk-Monitor einen 256 Byte großen Puffer benötigt. Dieser liegt von $BF00 bis $C000 im RAM unter dem BASIC, weil er dort am wenigsten stören kann.

READ: R (Track Sektor)

Liest einen Block von der Diskette in den Computer. Track und Sektor müssen als Hexzahlen eingegeben werden. Die erste Zeile des Blocks wird ausgegeben. Da wir dazu normale SMON-Routinen verwenden, steht als Speicheradresse $BF00. Das »BF« können Sie vorerst ignorieren.

Die weitere Ausgabe des Hexdump erfolgt anders als gewohnt mit der Taste <SHIFT>. <STOP> bricht die Ausgabe ab. Sie können die Hexbytes überschreiben und damit ändern. Eine dauerhafte Änderung erfolgt aber erst beim Zurückschreiben auf die Diskette

(siehe Befehl »W«). Geben Sie nur »R« ohne Track und Sektor ein, wird der logisch (!) nächste Block eingelesen.

MEMORY-DUMP: M
Zeigt den gerade im Puffer befindlichen Block nochmals auf dem Bildschirm an. Genau wie beim R-Befehl können Sie die Ausgabe mit <SHIFT> und <STOP> steuern und Änderungen vornehmen.

WRITE: W (Track Sektor)
Schreibt einen Block aus dem Puffer auf die Diskette zurück. Ähnlich wie bei »R« kann die Angabe von Track und Sektor entfallen. Es werden dann Track und Sektor des letzten R-Befehls benutzt. Das ist in fast allen Fällen auch der Richtige.

ERROR: @
Liest den Fehlerkanal aus, gibt ihn aber nur aus, wenn wirklich ein Fehler vorhanden war. (»00, OK, 00, 00 wird unterdrückt.)

EXIT: X
Verlässt den Disk-Monitor und springt in den SMON zurück. Dabei wird die Rahmenfarbe auf Blau zurückgeschaltet und es erscheint wieder der».« am Anfang der Zeile. Das BASIC wird wieder eingeschaltet. Wollen Sie nun mit SMON-Kommandos auf den Puffer zugreifen, müssen Sie BASIC wieder abschalten ($36 in Speicherstelle $0001).

Die folgenden Beispiele sollen Ihnen die Arbeit mit dem Disk-Monitor verdeutlichen. Achtung! Benutzen Sie unbedingt zum Üben eine Diskette, die Sie nicht mehr brauchen! Weder wir noch der Verlag haften dafür, wenn Ihr Lieblingsprogramm oder die mühsam erstellte Adressdatei unwiederbringlich dahin sind. Dass das sehr sehr schnell gehen kann, wissen wir aus eigener Erfahrung ...

Am Besten machen Sie von einer Ihrer Disketten eine Kopie, die Sie zum Üben benutzen können.

Reparatur eines gelöschten Files
Sicher ist Ihnen das auch schon passiert: Sie wollen Ihr Programm mit Namen »Schrott« löschen, geben als Abkürzung »S:S*« ein und merken in dem Moment, in dem Sie <RETURN> drücken, dass auf der Diskette auch alle Versionen von »SMON« waren, außerdem auch noch »Springvogel«, »Soccer« etc. Verzweifeln müssen Sie nur, wenn auch diese letzte SMON-Version mit dem Disk-Monitor dabei war. Ansonsten behalten Sie die Ruhe und verfahren Sie wie im Folgenden beschrieben.

Laden Sie also jetzt SMON, legen Sie Ihre »Übungsdiskette« (!) ins Laufwerk und löschen Sie eins der ersten Programme mit dem üblichen Scratch-Kommando.

Nun starten Sie SMON und drücken <Z>. Der Bildschirm ändert seine Farbe wie beschrieben, und am Anfang der Zeile erscheint der »*«. Jetzt geben Sie ein: R1200

Auf dem Bildschirm erscheint die erste Zeile der BAM, die bei jeder Diskette auf Track 18, Sektor 0 abgelegt ist. Die ersten beiden Bytes enthalten »12 01« und geben damit den logisch nächsten Block an.

In diesem Fall wäre das der erste Block des Directorys. Wenn Sie mit <SHIFT> die Bildschirmausgabe fortsetzen, erkennen Sie etwa in der Mitte den Diskettennamen. Lassen Sie die Ausgabe durchlaufen, bis wieder der »*« erscheint. Nun geben Sie »R« ohne weitere Angaben ein. Damit erhalten Sie den Koppel-Block, also Track 18, Sektor 1, den ersten Directory-Block. (Natürlich hätten Sie auch gleich »R 12 01« eintippen können, aber wir wollen ja zeigen, wie die Befehle funktionieren.)

In diesem Block stehen die ersten acht Programme Ihrer Übungsdiskette, auch der Name des soeben gelöschten ist dabei.

Trotzdem ist dieses Programm tatsächlich gelöscht und erscheint nicht mehr, wenn Sie sich das Directory anzeigen lassen. Vergleichen Sie den Eintrag des gelöschten Programms mit den anderen, fällt auf, dass 3 Bytes vor Beginn des Namens bei allen an-

deren »82« steht (sofern es sich um Programmfiles handelt), bei dem gelöschten aber »00«. Die Reparatur ist nun denkbar einfach: Sie brauchen lediglich die »00« mit »82« zu überschreiben.

Einen Haken hat die Sache allerdings noch: Beim Scratchen sind die vom Programm belegten Blöcke in der BAM als frei gekennzeichnet worden, und jeder neue Eintrag würde das als gelöscht gekennzeichnete File endgültig überschreiben. Um das zu verhindern, müssen Sie nach erfolgter Reparatur die Diskette validieren (von BASIC aus mit Kommando: `OPEN 1,8,15,"V"`). Dabei wird die BAM neu erzeugt und korrigiert.

Schützen eines Files

Da wir gerade dabei sind, wollen wir unser repariertes gelöschtes File gleich ein für alle Mal gegen Löschen schützen. Diese Möglichkeit des Diskettenoperationssystems (DOS) ist zwar nicht im Handbuch beschrieben, funktioniert aber trotzdem ausgezeichnet. Laden Sie dazu nochmals die erste Seite des Directorys mit `R 12 01` und ändern Sie die »82« vor dem Fileeintrag in »C2«.

Geben Sie »W« ein, um die Änderung auf Diskette zu schreiben. Verlassen Sie nun SMON mit »X« und lassen Sie sich ein Directory anzeigen. Das geschützte File ist mit einem »>« gekennzeichnet.

Versuchen Sie nun, dieses Programm mit dem Scratch-Kommando zu löschen. Es geht nicht! Zum »Entriegeln« brauchen Sie nur das »C2« wieder in »82« zu ändern. Der »>« im Directory verschwindet, und das File ist nicht mehr geschützt.

Schützen einer Diskette

Wollen Sie eine ganze Diskette vor versehentlichem Löschen oder Formatieren schützen, gibt es die Möglichkeit, die Löschschutzkerbe abzukleben. Es geht jedoch auch anders. Achtung! Die im Folgenden beschriebene Prozedur lässt sich nicht mehr rückgängig machen, auch nicht mit dem Disk-Monitor! Nehmen Sie also eine Diskette, die Sie anschließend »hart formatieren« können (also mit Eingabe einer ID).

Starten Sie nun den Disk-Monitor und lesen Sie die BAM mit »`R 12 00`« ein. Das dritte Byte enthält »41«. Diese »41« ist ein Kennzeichen für das DOS der Floppys 1541 oder 4040. Ändern Sie diese Bytes durch Überschreiben in »45« und speichern Sie die Änderung mit »`W`« auf die Diskette zurück. Verlassen Sie nun SMON und versuchen Sie, etwas zu löschen. Ergebnis siehe oben.

Versuchen Sie auch, die Diskette »weich«, also zum Beispiel mit `OPEN 1,8,15,"N:TEST"` zu formatieren. Auch das ist jetzt nicht mehr möglich. Aber es kommt noch besser: Starten Sie noch einmal den Disk-Monitor und versuchen Sie, die Änderung durch Zurückschreiben der »41« an Stelle der »45« rückgängig zu machen. Auch das ist nicht mehr möglich - wir hatten Sie bereits gewarnt! Es bleibt lediglich die Möglichkeit, die Diskette »hart«, zum Beispiel mit `OPEN 1,8,15,"N:TEST"` zu formatieren.

Sollten Sie nun entgegen allen Warnungen doch Ihre Master-Diskette gegen Schreibzugriffe gesichert haben, verraten wir Ihnen ausnahmsweise, wie Sie den Eingriff trotzdem rückgängig machen können. Dazu überlisten wir das DOS des 1541-Laufwerkes, indem wir ihm vorgaukeln, es hätte eine Diskette im Normalformat vor sich. Wir verwenden den Memory-Write-Befehl, mit dem wir in die Speicherstelle 0101 (Zeropage-Adresse) des 1541-RAM einfach ein »A« schreiben. Der CHR$-Kode des »A« ist 65, oder in hexadezimaler Schreibweise 41.

Erinnern Sie sich? Dieser Wert stand ursprünglich im dritten Byte des Tracks 18, Sektor 0. Mit folgendem kleinen Programm umgehen wir einfach die DOS-Kennzeichnung, und wir können die Diskette wieder normal beschreiben. Am Sinnvollsten ist es, sofort den SMON zu starten, das vorher in 45 abgeänderte Byte wieder in 41 zu verwandeln und abzuspeichern. Die Diskette kann dann wieder zum Lesen und Schreiben

verwendet werden. Hier nun das kleine Programm:

```
10 OPEN 1,8,15
20 PRINT#1,"M-W"CHR$(1)CHR$(1)CHR$(1)CHR$(65)
30 CLOSE 1
```

Ändern des Diskettennamens oder der ID

Wir haben bereits oben gesehen, dass in Spur 18, Sektor 0 einer Diskette etwa in der Mitte der Diskettenname gespeichert wird. Dieser Name kann durch einfaches Überschreiben geändert werden; er darf bekanntlich bis zu 16 Zeichen enthalten.

Hat Ihr neuer Name weniger Buchstaben als der alte, müssen Sie die Lücken mit »A0« und nicht mit »20« als Leerzeichen ausfüllen. Dies gilt vor allem, wenn Sie mit dieser Methode Filenamen ändern wollen. Das geht natürlich im Prinzip genauso wie eben beschrieben.

Hinter dem Diskettennamen ist in Spur 18, Sektor 0 die ID abgelegt. Sie wird beim Formatieren vor jeden Sektor in einen so genannten Header geschrieben und dient dem DOS zur Identifikation der Diskette. Zusätzlich wird sie noch in der BAM gespeichert, damit sie beim Laden eines Directorys mit angezeigt werden kann.

Nun ist es grundsätzlich nicht möglich, die ID im Header eines Sektors ohne Formatieren zu ändern, wohl aber die Eintragung in der BAM und damit die ID, die im Directory angezeigt wird. Genau wie beim Namen ist dies durch einfaches Überschreiben in der BAM möglich.

Ändern eines Filetyps

Wenn Sie einmal versucht haben, ein sequentielles File, etwa eine Datei, mit LOAD zu laden, werden Sie gemerkt haben, dass dies nicht möglich ist. Das DOS behauptet einfach, ein solches File existiere nicht, und der Rechner meldet »FILE NOT FOUND«. Viele Spiele zum Beispiel legen die »Hall of Fame« oder die Highscore-Liste als sequentielle Datei ab. Mit dem Disk-Monitor ist es nun aber möglich, den Filetyp im Directory zu verändern. Erinnern Sie sich an die »82«, die im Directory vor jedem Filenamen steht? Bei sequenziellen Files steht dort »81«. Was zu tun ist, werden Sie sich denken können. Na klar, die »81« wird in »82« geändert, und schon ist die Datei ohne Weiteres ladbar - natürlich wieder erst nach dem Zurückschreiben mit »W«.

Sinnvoll ist dies natürlich nur von SMON aus (mit Eingabe einer Ladeadresse). Mit »M« oder »K« können Sie dann die Datei ansehen und natürlich auch ändern. Vergessen Sie nicht, die geänderte Datei nach dem Zurückschreiben wieder in ein sequenzielles File zu verwandeln. Verblüffen Sie Ihre Freunde doch mal mit einem auf diese Weise »errungenen« Highscore. Die Anerkennung Ihrer Umwelt ist Ihnen sicher.

Ändern der Startadresse eines Programms

Wir haben uns bisher auf Manipulationen in der BAM oder im Directory beschränkt. Wollen wir in einem Programm selbst Änderungen vornehmen, müssen wir etwas tiefer in die »Geheimnisse der Floppy« eindringen.

So ist es bisweilen interessant, die Startadresse eines Maschinenprogramms zu kennen oder zu ändern. Dazu gehen wir folgendermaßen vor:

Zunächst suchen wir mit »R 12 01« und eventuell weiteren Folgesektoren (1204, 1207 ...) den Fileeintrag im Directory. Die beiden Bytes hinter der »82« direkt vor dem Programmnamen geben an, in welcher Spur und in welchem Sektor das Programm startet.

Wenn dort zum Beispiel »0A 04« steht, beginnt das Programm in Spur 10, Sektor 4. Lesen Sie nun diesen Block mit »R 0A 04« ein. Die ersten beiden Bytes dieses Blocks zeigen auf den nächsten Block des Programms, die beiden nächsten Bytes enthalten die Startadresse in der üblichen Low-/High-Byte-Reihenfolge. Zum Ändern der Startadresse überschreiben Sie die Bytes mit der neuen und speichern den Block mit »W« auf die Diskette zurück.

Die Zusammenarbeit mit SMON

Mit all diesen Beispielen sind die Möglichkeiten des Disk-Monitors noch lange nicht erschöpft. Sie sollten Ihnen als Anregung für eigene Experimente dienen. Üben Sie aber unbedingt so lange, bis Sie alle Kommandos aus dem »FF« (oder dezimal 255) beherrschen.

Sie ersparen sich damit unnötigen Ärger und durchweinte Nächte. Besonders interessant ist es, von SMON aus auf den Puffer zuzugreifen und die SMON-Befehle auf den Puffer anzuwenden. Erwähnen möchte ich nur die Möglichkeit, Programme für das DOS direkt zu assemblieren und in einem bestimmten Sektor ablegen zu können, die Find-Routinen oder das »K«-Kommando für Textänderungen. Da der Puffer im RAM unter dem BASIC liegt, muss BASIC in solchen Fällen abgeschaltet werden. Ändern Sie dazu mit dem »M«-Befehl in Speicherstelle 0001 die »37« in »36«.

Haben Sie die Arbeit mit SMON beendet, können Sie mit »Z« in den Disk-Monitor schalten und den Pufferbereich mit »W« (Spur, Sektor) abspeichern.

Die Ausgabe von Diskettenfehlern

Beim Arbeiten mit dem Disk-Monitor werden sämtliche Fehler vom Laufwerk direkt, auch ohne Eingabe von »@«, ausgegeben, zum Beispiel »ILLEGAL TRACK OR SECTOR«, wenn Sie mit »R« einen Block lesen wollen, den es gar nicht gibt.

Einen Fehler hat das Programm allerdings, den wir nicht verschweigen wollen: Der letzte Block eines Files enthält als Koppeladresse »00 FF«. Da es einen solchen Block nicht geben kann, »weiß« das DOS, dass es am Ende angelangt ist. Versuchen Sie aber, den nächsten Block (Spur 0, Sektor 255!) mit »R« zu lesen, erscheint als Fehlermeldung nicht, wie es sein müsste, »ILLEGAL BLOCK OR SECTOR«, sondern »SYNTAX ERROR«. Das ist zwar eigentlich unerheblich, sollte aber erwähnt werden. Der Fehler liegt in der Routine, die unsere Zahleneingaben in das richtige Diskettenformat wandelt. Für eine »korrekte« Umwandlung fehlte einfach der Platz im Programm, wir mussten uns mit einer »Sparroutine« behelfen.

Abschließend noch ein SMON-Trick, den wir einem aufmerksamen Leser verdanken: Für eine Directory-Ausgabe fehlte der Platz im SMON. Es geht aber hilfsweise so: Laden Sie das Directory zum Beispiel mit `L "$" 8000` an einen freien Speicherplatz. Mit »M« oder »K« können Sie jetzt das Directory »lesen«. Damit sind alle wichtigen Funktionen für den Umgang mit der Diskette im SMON enthalten.

Noch ein Tipp an dieser Stelle: Wenn Sie SMON mit »X« verlassen, um ins BASIC zurückzuspringen, wird in aller Regel bereits Ihre erste Eingabe von BASIC aus mit einem »OUT OF MEMORY ERROR« quittiert.

Beim Laden von SMON setzt das Betriebssystem nämlich die Zeiger für das BASIC-Programmende auf das SMON-Ende, also irgendwo in den $C...-Bereich.

Damit ist aber jede weitere Eingabe von BASIC-Zeilen unmöglich und führt zu dem oben angegebenen Fehler. Das ist nebenbei auch der Grund, wenn der SMON-Befehl »B« zum Erzeugen von DATA-Zeilen einmal nicht funktionieren will. Abhilfe schafft ein simples »NEW« unmittelbar nach dem Laden von SMON.

SMON selbst bleibt dabei unbeeinträchtigt, denn er liegt ja nicht im BASIC-Bereich. SMON ist also jetzt ordnungsgemäß geladen und kann mit »SYS 49152« wie gewohnt gestartet werden.

Das war's. Wir glauben, wir haben am Anfang dieser Serie nicht zu viel versprochen: Wenn Sie alle fünf Folgen durchgearbeitet haben, liegt Ihnen mit SMON jetzt ein Maschinenmonitor mit wirklich außergewöhnlichen Leistungsmerkmalen vor, der wohl jedem Vergleich Stand hält.

Viele Leser haben SMON zum Einstieg in die Maschinensprache genutzt, und das war auch unser Ziel. Natürlich werden Sie, vor allem, wenn Sie noch Anfänger sind, nicht

sofort alle Möglichkeiten von SMON ausnutzen können; eine Weile werden Sie mit Disassembler und Memory-Dump vollauf beschäftigt sein.

Aber je tiefer Sie mit SMON in die Maschinensprache eindringen, desto mehr werden Sie dessen Fähigkeiten zu schätzen wissen.

Ohne gutes Werkzeug geht es nicht: SMON Folge 6

Erschienen in: 64'er 10/1985
Verantwortlicher Redakteur: Achim Hübner
Autor: Mark Richters

Diese Erweiterung stellt elf weitere Befehle zur Verfügung. So lässt sich der Monitor zum Beispiel frei im Speicher verschieben, und Sprites oder Zeichensätze können sehr einfach erstellt und geändert werden.

Um die Befehlserweiterung zu initialisieren, geht man folgendermaßen vor:

1. SMON absolut laden
2. Den BASIC-Lader (auf Diskette; siehe Vorwort) eintippen und abspeichern
3. Nach dem Start des Laders die Startadresse (dezimal) Ihrer SMON-Version eingeben, zum Beispiel 49152 (= $C000)
4. Den erweiterten SMON zum Beispiel mit "SMONEX" Startadresse Endadresse abspeichern

Die neuen Routinen werden, genau wie die meisten bereits vorhandenen, durch einen Buchstaben, zum Teil gefolgt von Adressenangaben, aufgerufen.

Bei den ersten drei Ausgabebefehlen kann der Speicherinhalt durch Überschreiben der Zeile geändert werden.

`Z 4000 (4100)` (Zeichendaten)
gibt den Speicherinhalt von $4000 (bis $40FF) folgendermaßen aus: Jeweils ein Byte pro Zeile wird in 8-Bit-Form dargestellt. Dabei ist ein »*« ein gesetztes, ein ».« dagegen ein nicht gesetztes Bit.

Die beiden Zeichen sind willkürlich gewählt und können durch Überschreiben der Speicherzellen $xE65, $xE2D (Bit = 1) und $xE69, $xE30 (Bit = 0) in den Bildschirmkode (!) der gewünschten Zeichen geändert werden.

Die Anwendung dieses Befehls liegt beispielsweise in der gezielten und anschaulichen Beeinflussung bestimmter Steuerbits in VIC, CIA, etc. Andererseits lassen sich - besonders in Verbindung mit dem Kommando Q - Zeichendaten leicht modifizieren.

`H 4000 (4100)`
entspricht dem Befehl Z mit dem Unterschied, dass jeweils drei Bytes pro Zeile ausgegeben werden. Das entspricht dem Format für Spritedaten. Auf diese Weise steht mit dem erweiterten SMON ein kleiner »Sprite-Editor« zur Verfügung.

`N 4000 (4100)` (Normaldarstellung)
interpretiert den Speicherinhalt von $4000 (bis $40FF) als Bildschirmkode und gibt 32 Zeichen pro Zeile aus.

`U 4000 (4100)` (Übersicht)
Wie N, jedoch werden in einer Zeile 40 Zeichen dargestellt. Änderungen sind nur mit N möglich. Dieser Befehl dient hauptsächlich dazu, im Speicher abgelegte Bildschirminformationen so auszugeben, wie sie tatsächlich im 40-Zeichen/Zeile-Format aussehen würden. Dieser Befehl ist recht nützlich, um professionelle Videospiele zu analysieren, da hier Spielszenen oft im Bildschirmkode gespeichert sind.

E 4000 (4100) (Erase)
ist der bereits in Folge 4 vorgeschlagene Erase-Befehl zum Füllen des Speicherbereichs von $4000 bis $40FF mit $00.

Y 40
kopiert die vorhandene SMON-Version in nur drei Sekunden nach $4000 bis $4FFF und nimmt dabei alle notwendigen Anpassungen vor. Die ursprüngliche Speicherversion des Monitors bleibt unverändert. Mit G 4000 kann man in den neuen SMON springen. Von dem Byte-Wert, der übergeben werden muss, wird nur das obere Nibble ($4) gewertet, so dass sich theoretisch 16 SMON-Versionen im Speicher unterbringen lassen, wobei natürlich nicht alle Möglichkeiten sinnvoll sind.

Auf diese Weise lässt sich stets die erforderliche Speicherversion herstellen, ohne dass langwierige Änderungen notwendig sind.

Q 2000
kopiert den Zeichensatz aus dem ROM von $D000 bis $DFFF in das RAM nach $2000. Dort kann er mit dem Befehl Z nach Belieben geändert werden. Möchte man zum Beispiel das Zeichen »A« in ein »Ä« umdefinieren, so ist der Zeichensatz mit Q 2000 ins RAM zu kopieren. Anschließend kann mit Z 2000 2015 der Bereich in binärer Form auf dem Bildschirm ausgegeben werden, in dem auch das Zeichen »A« steht. Dieses kann nun in ein »Ä« geändert werden, indem man mit dem Cursor an die zu ändernde Stelle fährt und für einen Punkt, der gesetzt werden soll, ein »*«, und für einen Punkt, der nicht gesetzt werden soll, ein ».« setzt.

So, jetzt ist der Zeichensatz umdefiniert, aber noch nicht aktiviert. Als Nächstes muss dem Videokontroller die Startadrese des neuen Zeichensatzes mitgeteilt werden. Dazu ist die Adresse $D018, in der eine hexadezimale 15 steht, durch eine hexadezimale 18 zu ersetzen.

J
bringt den letzten Ausgabebefehl (K, D, M, Z, H, N, U) auf den Bildschirm zurück. Mit <RETURN> wird der letzte Befehl noch einmal ausgeführt.

Zum Schluss noch ein Tipp:
DATA-Zeilen in Hex-Byte-Darstellung sind wegen ihrer konstanten Länge (immer zwei Ziffern pro Wert!) übersichtlicher als solche mit dezimalen Zahlen. Da für die Ausgabe von Hex-Werten bereits alle Routinen im SMON integriert sind, kann der B-Befehl (BASIC-DATA-Zeilen erzeugen) durch Verändern eines einzigen Sprungbefehls dahingehend manipuliert werden, dass der Speicherinhalt künftig in Form von Hex-Bytes ausgegeben wird:

Disassemblieren Sie dazu den Byte-Ausgabebefehl mit D x99F und ersetzen JSR BDD1 durch JSR x32A. Für das »x« muss der 4-KByte-Block, in dem die zu ändernde SMON-Version steht, eingesetzt werden. Liegt Ihre SMON-Version bei $C000, so ersetzen Sie das »x« durch ein »C«.

Die Gesamtlänge der DATA-Zeile kann außerdem durch Verändern der Speicherzelle $x9AE variiert werden. Bei dem Wert $1C werden zum Beispiel genau acht Hex-Byte pro Zeile ausgegeben.

Die Erweiterung finden Sie ebenso wie SMON selbst auf der Diskette, die unter www.skriptorium-vd.de/01/smon.d64 heruntergeladen werden kann.

Sie können auch eine 5,25-Zoll-Diskette bestellen (siehe Einleitung).

Index I – Assembler ist keine Alchemie

#
16-Bit-Division 108, 109, 111
16-Bit-Multiplikation 105
1-Byte-Befehle 20
2-Byte-Befehle 20
3-Byte-Befehle 20

A
Absturz 51
ADC 39, 40, 42, 46, 50, 60, 74, 80
Adressbus 15
Adressierungsarten 20
  absolut-X-indizierte Adressierung 59, 60
  absolut-Y-indizierte Adressierung 59, 60
  implizite Adressierung 22, 62
  indirekt-indizierte Adressierung 79
  indizierte Adressierung 59, 60
  indiziert-indirekte Adressierung 82
  relative Adressierung 47, 49
  Vektor-Adressierung 63
  Zeropage-Adressierung 49, 50, 51, 52, 59
  zeropage-absolute Adressierung 51
  zeropage-absolut-X-indizierte A. 60
  zeropage-absolut-Y-indizierte A. 60
AFAC 91, 92, 94, 113
Akku 23, 52, 62
Akkumulator. Siehe Akku
ALU 39
AND 61, 95, 96, 100
ARG. Siehe AFAC
arithmetische Reihe 42, 43, 44
arithmetisch-logische Einheit. Siehe ALU
ASCII 20, 53, 54, 55, 115
  Commodore-ASCII 53, 54, 55
ASL 98, 100, 101, 102
Assembler 12, 14

B
BCC 45, 46, 60
BCD. Siehe Zahlensysteme
BCS 45, 46, 60
BEQ 45, 46, 60
Binärzahlen. Siehe Zahlensysteme
BIT 60, 61, 74
BMI 45, 46, 60
BNE 33, 34, 45, 46, 47, 60
BPL 45, 46, 60
Branch-Befehle 45
BRK 21, 119, 121, 122, 126, 129
BVC 45, 46, 60
BVS 45, 46, 60

C
Carry-Bit. Siehe Flaggen
CHR$-Kodes 58
CIAs 124, 129, 136, 139, 143, 146, 147
  CIA1 137, 139, 147CIA 2 137, 147
  Timer 144, 147
  Timer A 144, 145, 147
  Timer B 145
CLC 39, 41, 42, 46, 60
CLD 30, 34, 40, 60
CLI 119, 120, 126
CLV 60, 61, 74
CMP 50, 52, 60, 74, 80
Computer, Aufbauprinzip 15
CPU. Siehe Prozessor
CPX 50, 52, 60
CPY 50, 52, 60
CRA 145, 148
CRB 145, 148, 149

D
Datenbus 15
DEC 27, 28, 34, 40, 50, 60, 74
DEX 27, 28, 29, 34, 40, 60
DEY 27, 28, 34, 40, 60
Dezimalzahlen. Siehe Zahlensysteme

E
EOR 97, 100, 126

F
FAC 67, 92, 94
Fehlermeldungen 68
  OUT OF MEMORY ERROR 76
  OVERFLOW ERROR 103
  SYNTAX ERROR 68

Fehlernummern 83
Flaggen 29
  B-Flagge 121
  Carry-Flagge 37, 38, 39, 41, 45, 52, 102
  C-Flagge. Siehe Carry-Flagge
  D-Flagge 40
  N-Flagge 29, 45, 52, 61, 62
  V-Flagge 37, 38, 39, 45, 46, 61
  Z-Flagge 29, 40, 45, 52, 62, 102
Fließkomma-Akkumulator 1. Siehe FAC
Fließkomma-Akkumulator 2. Siehe AFAC
Fließkommazahlen 65, 91
FLPT. Siehe FAC

H
Hexadezimalzahlen. Siehe Zahlensysteme

I
ICR 149
INC 27, 28, 29, 34, 40, 50, 60, 74
Index-Register 16, 59
  X-Register 16, 23, 28, 52, 59, 62
  Y-Register 16, 23, 28, 52, 59, 60, 62
Interrupts 117, 119, 122, 123, 124, 136, 138, 143
Interrupt-Vektor 19
INX 27, 28, 29, 34, 40, 60
INY 27, 28, 34, 40, 60
IRQ 19, 119, 120, 121, 122, 127, 128, 129, 137, 139, 141, 147, 149
IRQ-CIA 125
IRQ-CIA 143

J
JMP 60, 62, 74, 121
Joystickports 104
JSR 60, 62, 65, 74

K
Kernal-Adressen 70
Kernal-Routinen 55, 69, 82
  BLTUC 110, 112, 113
  CHKOUT 85
  CHRGET 55, 56, 57, 58
  CHRGOT 58
  CHROUT 84
  CLOSE 86
  CLRCHN 86
  FFD2 82
  MOVAF 95
  MOVMF 95
  OPEN 85
  SETLFS 85
  SETNAM 86
  UMULT 106, 107
Komplemente 31
  Einerkomplement 31
  Zweierkomplement 31, 38, 39

L
LDA 20, 23, 24, 50, 60, 74, 80
LDX 22, 23, 24, 50, 60, 74
LDY 22, 23, 24, 50, 60, 74
logische Verknüpfungen 61
LSR 101, 102, 105, 116

M
Maschinensprache 12
MFLPT. Siehe AFAC
Module 135
  Modulsimulation 136
  Modulstart 135

N
Nibble 24
NMI 122, 125, 126, 130, 131, 132, 149
NMI-CIA 125, 143
NOP 60, 62, 74
Normalisieren 90

O
Offset 48, 49
ORA 96, 100

P
Paritätsbit 53
PHA 76, 80
PHP 76, 77, 80
PLA 76, 80
PLP 76, 77, 80
Programmzähler 16
  PCH 16
  PCL 16

Prozessor 12, 15, 118, 122
  Aufbauschema 16
Prozessorport 78
Prozessorstack. Siehe Stapelregister
Prozessorstatusregister 19

R
RAM 15, 70, 79
RESET 132, 133, 136
RESTORE-Taste 125, 131, 132, 136, 148
ROL 102, 103, 116
ROM 15, 68, 72, 79, 110, 113
ROR 102, 103, 104, 105, 116
RS232C-Schnittstelle 143
RTI 119, 120, 121, 126
RTS 14, 22, 23, 24, 60
RUN/STOP-Taste 131, 132, 148

S
SBC 41, 42, 46, 50, 60, 74, 80
SEC 41, 42, 46, 60
SED 30, 34, 60
SEI 119, 120, 126
Speicheraufbau 17
  C 64 17, 68
  VC 20 17-18
Sprungweite. Siehe Offset
STA 21, 23, 24, 50, 60, 74, 80
Stackpointer 16
Stapelregister 16, 63, 75, 76, 78
Stapelzeiger 63
STX 22, 23, 24, 50, 60, 74
STY 22, 23, 24, 50, 60, 74

T
Taktfrequenz 17
TAX 60, 62, 74
TAY 60, 62, 74
TI$. Siehe Uhren, interne
Timer. Siehe CIAs
TSX 76, 77, 78, 80
TXA 60, 62, 74
TXS 76, 77, 78, 80
TYA 60, 62, 74

U
Uhren, interne 15, 124, 129, 144
  Echtzeituhren 147
Unterbrechungen. Siehe Interrupts
USR 67, 68, 70, 72

V
Variablen 31, 33
Vektoren 63, 120, 128, 129
Vergleichsbefehle 53
Verzögerung 33, 34, 35
VIC-II 122, 129, 137, 138, 139

W
Wahrheitstabellen 61, 96, 97

Z
Zahlensysteme 20
  Binärzahlen 20, 89
  Binärzahlen, negative 30
  Signed-Binary-Binärzahlen 30
  Dezimalzahlen 20
  binär kodierte Dezimalzahlen 29, 30, 148
  Hexadezimalzahlen 24
Zeichen-ROM 54
Zeropage 49

Index II – Ohne gutes Werkzeug geht es nicht: SMON

$ 160
% 160
? 160
16-Bit-Vergleich 163, 169

A
A [ANFADR] 154
ADD 160
Ändern 184
  Ändern der Startadresse eines Programms 184
  Ändern eines Filetyps 184
  Änderns des Diskettennamens oder der ID 184
ASCHEX 156
Assemblieren 154

B
B (ANFADR ENDADR) 165
BASIC-DATA 165
Befehlserweiterung 187
Befehlsübersicht 178

C
C (ANFADRaltENDADRalt ANFADRneu ANFADRges ENDADRges) 161
CHRIN 156
CMDEXEC 157
CMDFOUND 157
CMDS 157
CMDSEARCH 157
CMDTBL 157
COMMAND 157
CONVERTIEREN (Verschieben eines Programms mit Adressumrechnung) 161

D
D [ANFADR,ENDADR] 155
Disassemblieren 155
Diskette, Schützen einer D. 183
Diskettenmonitor 181
  Ausgabe von Diskettenfehlern 185
  Befehle des Disk-Monitors 181
  Zusammenarbeit mit SMON 185

E
E 188
ENDHI 163
ENDLO 163
Erase 188
ERROR: @ 182
EXECUTE 157
Exit 156
EXIT: X 182

F
F (HEX-WERT(e), ANFADR ENDADR) 166
Files 182
  Reparatur eines gelöschten Files 182
  Schützen eines Files 183
FIND 166
FIND-Kommandos 168

G
G [ADRESSE] 155
GET2ADR 157
GET3ADR 157
GETADR 156, 157
GETBYT 156
GETCHRERR 156, 157
GETRET 157, 163
GETSTART 157

H
H 187
Hexadezimalsystem 160, 188

I
I 01 159
I/O-SET 159

J
J 188

K
K (ANFADR ENDADR) 166
Kernal 163
Kernal-Einsprünge 163
Kernal-Routinen 163

Kommando-Liste 175
Kontrolle 166

L
L "Name" 159
LOAD 159

M
M [ANFADR ENDADR] 155
Memory-Dump 155
MEMORY-DUMP: M 182

N
N 187

O
O (ANFADR ENDADR HEX-Wert) 161
OCCUPY (Besetzen) 161

P
P 02 160
PCH 157, 163
PCL 157, 163
PRINTER-SET 160

Q
Q 188

R
R 155
READ: R (Track Sektor) 181
Registeranzeige 155
RTS 157

S
S "Name", ANFADR ENDADR 160
SAVE 160
SMON verschieben 176
SMON-Speicherstellen 176
Starten eines Maschinenprogramms (Go) 155
SUB 160

T
TB (Adresse Anzahl-der-Durchläufe) 174
TQ (Adresse) 174
Trace Break 174
Trace Quick 174
Trace Walk 172
Trace-Stopp 171
TS (Startadresse Stoppadresse) 171
TW (Startadresse) 172

U
U 187
Übersicht. Siehe Befehlsübersicht
Umrechnungen 160
  binär-dez 160
  binär-hex 160
  dez-hex 160
  hex-dez 160

V
V (ANFADRalt ENDADRalt ANFADRneu ANFADR ENDADR) 161
VARIATION 161

W
W(ANFADRalt ENDADRalt ANFADRneu) 161
WRITE 161
WRITE: W (Track Sektor) 182

X
X 156

Y
Y 188

Z
Z 181, 187

Abbildungsverzeichnis

Abb. 01 Aufbauprinzip eines 8-Bit-Computers 15
Abb. 02 Aufbauschema eines 6510-Prozessors 16
Abb. 03 VC 20-Speicher (Grundversion oder mit 3-KB-Erweiterung) 17
Abb. 04 VC 20-Speicher (Version mit mehr als 6,5 KB Speicherplatz 18
Abb. 05 Der Monitor zeigt das nackte Programm im Speicher. 32
Abb. 06 So werden Integer-Variablen aus BASIC-Programmen vom C64 im Speicher eingerichtet. 33
Abb. 07 Flussdiagramm zur Verzögerungsschleife 35
Abb. 08 Das Carry-Bit als Bit 8 einer Rechenoperation 38
Abb. 09 Flussdiagramm des Assemblerprogramms »Summe einer arithmetischen Reihe« 43
Abb. 10 Flaggen bei den Vergleichsbefehlen 53
Abb. 11 Grafikzeichen zu den entsprechenden CHR$-Kodes 58
Abb. 12 Das Flussdiagramm zu dem im Text erläuterten Programm 71
Abb. 13 So wird der Stapel gefüllt. 75
Abb. 14 Der Aktenstapel 76
Abb. 15 Die Abfolge der Routineaufrufe 87
Abb. 16 Wirkung des ASL-Befehls - arithmetisches Linksschieben 98
Abb. 17 Wirkung von LSR auf ein Byte 101
Abb. 18 Wirkung von ROL auf ein Byte 102
Abb. 19 Wirkung der Befehlskombination ASL(-) und ROL(-) auf zwei Bytes (LSB/MSB) 103
Abb. 20 Wirkung von ROR auf ein Byte 103
Abb. 21 Flussdiagramm zur Betriebssystemroutine UMULT 106
Abb. 22 Flussdiagramm des Programms zur 16-Bit-Division 109
Abb. 23 Flussdiagramm zur Betriebssystemroutine BLTUC 112
Abb. 24 Unterbrechungen im täglichen Leben 118
Abb. 25 Die CPU rettet den Programmzähler und das Status-Register beim Eintreten einer Unterbrechung auf den Stapel 119
Abb. 26 Das Interrupt-Enable-Register des VIC-II-Chips 123
Abb. 27 Das Interrupt-Latch-Register des VIC-II-Chips 123
Abb. 28 Genereller Aufbau der Unterbrechungs-Kontroll-Register der beiden CIA-Bausteine 124
Abb. 29 So bearbeitet der C64 normale IRQs. 128
Abb. 30 Auf diese Weise verläuft ein unvorhergesehener BRK im Sande. 129
Abb. 31 Flussdiagramm zum Ablauf einer NMI-Unterbrechung 130
Abb. 32 Diesem Weg folgt ein RESET. 132
Abb. 33 Diesen Inhalt müssen die Speicherstellen $8000 bis $8008 haben, damit ein Modulstart stattfindet. 135
Abb. 34 Flussdiagramm zu Programm 8 (Modulsimulation) 136
Abb. 35 So sieht das 9-Bit-Register im VIC-II-Chip aus, das die Rasterzeilen mitzählt. 138
Abb. 36 Flussdiagramm zu Programm 9. Ein bunter Rahmen wird erzeugt. 140
Abb. 37 Das Kontrollregister von Timer A 145
Abb. 38 Das Kontrollregister von Timer B 145
Abb. 39 Die Register der Echtzeituhren 147

Tabellenverzeichnis

Tab. 01 BASIC-Start- und Endadressen beim VC 20 mit unterschiedlichem Speicherausbau 18
Tab. 02 eine Registeranzeige 19
Tab. 03 das Prozessor-Statusregister P - die Flaggen 19
Tab. 04 Folge 2: Befehlsübersicht 24
Tab. 05 Vergleich von Dezimal- und Binärzahlen 29
Tab. 06 Folge 3: Befehlsübersicht 34
Tab. 07 Folge 4: Befehlsübersicht 46
Tab. 08 Folge 5: Befehlsübersicht 50
Tab. 09 ASCII- und Commodore-ASCII-Kode 54
Tab. 10 Bedeutung der Abkürzungen im ASCII-Kode 55
Tab. 11 Anwendbarkeit der indizierten Adressierungsarten auf die bisher gelernten Assembler-Befehle 60
Tab. 12 Wahrheitstabelle der logischen Verknüpfung UND 61
Tab. 13 Kernal-Routinen 69
Tab. 14 Folge 6: Befehlsübersicht 74
Tab. 15 Die Tabelle zeigt, welche Bausteine bei verschiedener Belegung der Bits Bits 0 bis 2 des Prozessorports (Speicherstelle 1) eingeschaltet sind. 78
Tab. 16 Folge 7: Befehlsübersicht 80
Tab. 17 Fehlernummern und ihre Bedeutung. Die Nummern findet man bei gesetztem Carry im Akku. 83
Tab. 18 So sieht die Zahl 1985,125 komplex in FAC und ARG aus. 91
Tab. 19 Wahrheitstabelle zur AND-Verknüpfung 96
Tab. 20 Wahrheitstabelle zur ORA-Verknüpfung 96
Tab. 21 Wahrheitstabelle zur EOR-Verknüpfung 97
Tab. 22 Folge 8: Befehlsübersicht 100
Tab. 23 UMULT am Beispiel der Multiplikation 321 * 65 = 20865 107
Tab. 24 16-Bit-Division Schritt für Schritt am Beispiel 20867 : 321 = 65 Rest 2 111
Tab. 25 Im ROM stehen nicht nur Programme, sondern auch Tabellen. Hier einige wichtige Zahlen 114
Tab. 26 Diese Texte sind im ROM als ASCII-Werte abgelegt. 115
Tab. 27 Folge 9: Befehlsübersicht 116
Tab. 28 Unterbrechungsvektoren und ihre Inhalte 120
Tab. 29 Folge 11: Befehlsübersicht 126
Tab. 30 Die wichtigen Register der beiden CIAs 143

Programmverzeichnis

Progr. 1 Test und Demonstration der Verschieberoutine. Das Programm zeigt das Ein- und Ausschalten einer Kopfzeile auf dem Bildschirm. 73
Progr 2 Testprogramm für die beiden kleinen Assembler-Routinen. 93
Progr. 3 Das Demo-Programm zur neuen Verschieberoutine. Vorher müssen Programm 4 und Programm 5 geladen werden. 113
Progr. 4 Die 16-Bit-Division 115
Progr. 5 Zweiter Teil der Verschieberoutine 116
Progr. 6 Verschieberoutine 115
Progr. 7 Simulation eines Moduls 135
Progr. 8 Das im Artikel entwickelte Programm auf einen Blick 141
Progr. 9 »Timer-Test«, ein Beispiel für die Anwendung eines 32-Bit-Timers 150
Progr. 10 Eine Echtzeituhr 150

Tabellenverzeichnis

Tab. 01 BASIC-Start- und Endadressen beim VC 20 mit unterschiedlichem Speicherausbau 18
Tab. 02 eine Registeranzeige 19
Tab. 03 das Prozessor-Statusregister P - die Flaggen 19
Tab. 04 Folge 2: Befehlsübersicht 24
Tab. 05 Vergleich von Dezimal- und Binärzahlen 29
Tab. 06 Folge 3: Befehlsübersicht 34
Tab. 07 Folge 4: Befehlsübersicht 46
Tab. 08 Folge 5: Befehlsübersicht 50
Tab. 09 ASCII- und Commodore-ASCII-Kode 54
Tab. 10 Bedeutung der Abkürzungen im ASCII-Kode 55
Tab. 11 Anwendbarkeit der indizierten Adressierungsarten auf die bisher gelernten Assembler-Befehle 60
Tab. 12 Wahrheitstabelle der logischen Verknüpfung UND 61
Tab. 13 Kernal-Routinen 69
Tab. 14 Folge 6: Befehlsübersicht 74
Tab. 15 Die Tabelle zeigt, welche Bausteine bei verschiedener Belegung der Bits Bits 0 bis 2 des Prozessorports (Speicherstelle 1) eingeschaltet sind. 78
Tab. 16 Folge 7: Befehlsübersicht 80
Tab. 17 Fehlernummern und ihre Bedeutung. Die Nummern findet man bei gesetztem Carry im Akku. 83
Tab. 18 So sieht die Zahl 1985,125 komplex in FAC und ARG aus. 91
Tab. 19 Wahrheitstabelle zur AND-Verknüpfung 96
Tab. 20 Wahrheitstabelle zur ORA-Verknüpfung 96
Tab. 21 Wahrheitstabelle zur EOR-Verknüpfung 97
Tab. 22 Folge 8: Befehlsübersicht 100
Tab. 23 UMULT am Beispiel der Multiplikation 321 * 65 = 20865 107
Tab. 24 16-Bit-Division Schritt für Schritt am Beispiel 20867 : 321 = 65 Rest 2 111
Tab. 25 Im ROM stehen nicht nur Programme, sondern auch Tabellen. Hier einige wichtige Zahlen 114
Tab. 26 Diese Texte sind im ROM als ASCII-Werte abgelegt. 115
Tab. 27 Folge 9: Befehlsübersicht 116
Tab. 28 Unterbrechungsvektoren und ihre Inhalte 120
Tab. 29 Folge 11: Befehlsübersicht 126
Tab. 30 Die wichtigen Register der beiden CIAs 143

Programmverzeichnis

Progr. 1 Test und Demonstration der Verschieberoutine. Das Programm zeigt das Ein- und Ausschalten einer Kopfzeile auf dem Bildschirm. 73
Progr 2 Testprogramm für die beiden kleinen Assembler-Routinen. 93
Progr. 3 Das Demo-Programm zur neuen Verschieberoutine. Vorher müssen Programm 4 und Programm 5 geladen werden. 113
Progr. 4 Die 16-Bit-Division 115
Progr. 5 Zweiter Teil der Verschieberoutine 116
Progr. 6 Verschieberoutine 115
Progr. 7 Simulation eines Moduls 135
Progr. 8 Das im Artikel entwickelte Programm auf einen Blick 141
Progr. 9 »Timer-Test«, ein Beispiel für die Anwendung eines 32-Bit-Timers 150
Progr. 10 Eine Echtzeituhr 150

64'er

mit SMON
Anleitung + Programm

Jeder Programmierer will die volle Leistung aus seinem Rechner herausholen. Bei Computern mit Acht-Bit-Prozessoren stößt man jedoch schnell an deren Geschwindigkeitsgrenzen. Daher sind für Programmierer zwei Dinge unverzichtbar:

- **ein sauberer und möglichst Speicherplatz sparender Programmierstil und**
- **die Benutzung einer Programmiersprache, die der Prozessor schnell versteht.**

Die Programmiersprache der Wahl heißt deshalb »Assembler«.
Dieses Buch wendet sich an alle, denen das eingebaute BASIC des VC20 oder C64 zu langsam geworden ist – immerhin ist Assembler ungefähr 1000-mal schneller.

Dieser Kurs ist übrigens für alle Computer geeignet, die den Prozessor 6502 oder 6510 besitzen. Das sind neben Commodore-Rechnern auch Modelle von Apple, Atari und einigen anderen Herstellern.

Zusätzlich zum 64'er-Kurs »Assembler ist keine Alchemie« finden Sie im hinteren Teil des Buches noch die Bedienungsanleitung zu SMON, einem sehr leistungsfähigen und ausgereiften Assembler für den C64. Das Programm selbst und viele der angesprochenen Beispielprogramme können Sie auf der Webseite des Verlags als D64-Dateien herunterladen. Wenn Sie auf einem »echten« C64 programmieren und keine Möglichkeit zum Konvertieren der Datei auf eine 5,25-Zoll-Diskette haben, können Sie eine beim Verlag anfordern.

ISBN 3-938199-01-6

24,80 EUR (Deutschland)
25,50 EUR (Österreich)
38,00 SFR (Schweiz)

skriptorium